UNIVERSITÉ MONTPELLIER 2
SCIENCES ET TECHNIQUES

2012

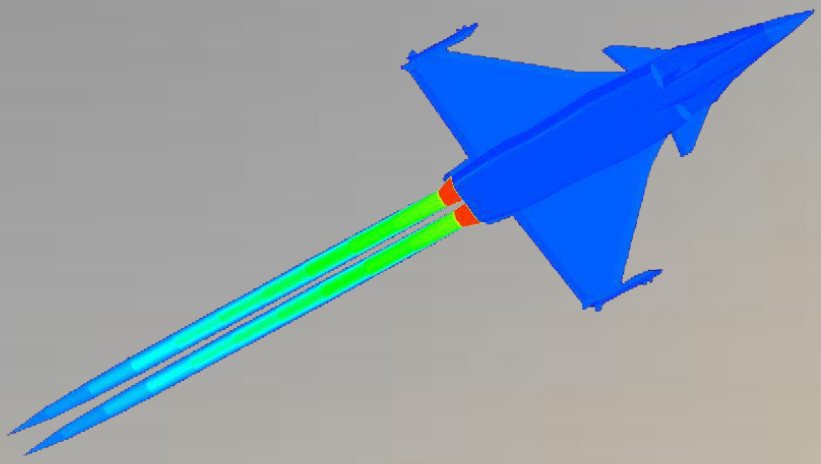
ONERA
THE FRENCH AEROSPACE LAB

Rapport de Stage

Crira / Advanced Rendering Tools

Réalisation d'une solution de rendu graphique sous GPU

Amélioration d'un modèle thermodynamique



Master Physique et Informatique

Emmanuel Hermellin

Informations complémentaires sur le rapport de stage :

Si vous voulez utiliser ou vous faire une idée des performances de ARTGpu, vous pouvez l'installer sur votre machine la version utilisateur (il faut une carte Nvidia compatible Cuda (de type Fermi minimum), nécessite MinGW) ou la version développeur (il faut une carte Nvidia compatible Cuda (architecture Fermi), avoir installé Cuda Toolkit et Optix SDK ainsi que Visual Studio Express, Cmake et enfin MinGW).

Lien de téléchargement ARTGpu version utilisateur :
https://www.dropbox.com/s/4r79rvbpl134v8e/ARTGPU_Version_Utilisateur.rar

Lien de téléchargement ARTGpu version développeur :
https://www.dropbox.com/s/gsehqq59idnkofx/ARTGPU_Version_Developpeur.rar

Pour vous aider dans l'installation et l'utilisation, vous trouverez en annexe un manuel d'utilisation de ce logiciel.

Sommaire

I. PRESENTATION 8

1. Le stage et son rapport 8
2. L'Onera, The French Aerospace Lab (article issu du site de l'Onera) 9
3. Le DOTA 10
4. Crira 10
5. Le Projet 11

II. INTRODUCTION AUX NOTIONS ESSENTIELLES 12

1. Raytracing 12
2. Path Tracing et Photon Mapping 13
3. Les structures accelératrices 14
4. Les structures accelératrices et le raytracing 16
5. La programmation sous GPU : Cuda 17
6. Optix, Raytracing Engine 19

III. PROBLEMATIQUES ET SOLUTIONS 20

1. La situation actuelle 20
2. Les possibilites 20
3. Les choix 21

IV. DEVELOPPEMENT ET PROGRAMMATION 22

1. Cahier des charges 22
2. Decoupage des projets 23
3. Developpement du logiciel Advanced Rendering Tools Gpu (ARTGpu) 23
   a. *Mise en place du Raytracing 23*
   b. *Les tests de performances du Raytracing 27*
   c. *Les améliorations 30*
   d. *Mise en place du Photon Mapping 32*
   e. *Mise en place du Path Tracing Markov Chain Monte Carlo 34*
   f. *Diagrammes du projet ARTGpu 36*
   g. *Fonctions communes, problèmes et solutions 38*
4. Compatibilite 38
5. Test de performance ARTGpu 39
6. Integration avec Crira 39
7. Ouverture et evolutions futures 39

V. LES METHODES DE RENDU EXPLIQUEES PAR LA PHYSIQUE 40

1. L'equation de rendu (article informatif issu d'internet) 40
2. La physique du raytracing (article informatif issu d'internet) 41
   a. *Le modèle de Phong 42*
   b. *Le modèle de Fresnel 43*
3. La physique du Photon Mapping 44
4. La physique du Path tracing et MCMC 46
5. Comparatifs, avantages, defauts et roles des differents modules 48
   a. *Avantages et défauts 48*
   b. *Comparatifs des modules 49*

VI. MODELE THERMIQUE APPLIQUE AUX TUYERES DANS CRIRA 52

1. LE MODELE THERMIQUE ACTUEL 52
2. LES TURBOREACTEURS 52
3. ETUDE THERMODYNAMIQUE DE LA TUYERE 55
   a. *Calcul des principaux paramètres d'une tuyère convergente* 55
   b. *Cas des turboréacteurs double flux* 57
   c. *Température de la tuyère* 57
   d. *Programmation du modèle* 58

VII. CONCLUSION 60

VIII. BIBLIOGRAPHIE / WEBOGRAPHIE 62

IX. ANNEXES 64

1. LA PROGRAMMATION AVEC CUDA 64
   a. *Introduction* 64
   b. *Notions de threads, grille et blocs* 64
   c. *Les fonctions, variables et routines principales* 65
2. LA PROGRAMMATION AVEC OPTIX 67
3. LES DIFFERENTS TYPES DE FICHIERS 70
   a. *Les fichiers OBJ (article issu de wikipédia)* 70
   b. *Les fichiers BDD (article technique de l'Onera)* 71
4. MANUEL D'UTILISATION ARTGPU 72
   a. *Installation version développeur* 72
   b. *Installation version utilisateur* 72
   c. *Fonctionnement détaillé de ARTGpu* 73
   d. *Historique des versions* 76
5. DIAGRAMME DE GANTT 77

Glossaire

CUDA : *Compute Unified Device Architecture* ou architecture matérielle unifiée de calcul. C'est une bibliothèque de calcul permettant l'utilisation du matériel de rendu graphique construit par NVIDIA à d'autre fin que l'affichage direct.

OpenCL : *Open Computing Language* est la combinaison d'une API et d'un langage de programmation dérivé du C, proposé comme un standard ouvert par le Khronos Group. OpenCL est conçu pour programmer des systèmes parallèles hétérogènes comprenant par exemple à la fois un CPU multi-cœur et un GPU.

OpenGL : *Open Graphics Library* est une spécification qui définit une API multiplateforme pour la conception d'applications générant des images 3D (mais également 2D). L'interface regroupe environ 250 fonctions différentes qui peuvent être utilisées pour afficher des scènes tridimensionnelles complexes à partir de simples primitives géométriques.

GPU : *Graphical Processing Unit* ou Unité de calcul graphique. Représentée dans l'ordinateur par la pièce connue sous le nom de carte graphique.

GPGPU : *Global Purpose GPU.* Type particulier de GPU, cette unité de calcul se comporte à présent comme un coprocesseur à part entière.

Raytracing: Le **lancer de rayon** (anglais: **ray tracing**) est une technique de rendu en synthèse d'image simulant le parcours inverse de la lumière : on calcule les éclairages de la caméra vers les objets, puis vers les lumières, alors que la lumière va de la scène vers l'œil.

Path Tracing: Le **path tracing** est une technique de lancer de rayon (**ray tracing**), utilisée pour déterminer l'illumination globale d'une scène 3D, pouvant aller jusqu'à générer l'image finale par une constitution progressive. L'image n'est d'abord qu'un brouillard de pixels qui s'affine progressivement jusqu'à être débarrassée presque complètement de son "grain".

Photon Mapping : le **photon mapping** est un algorithme d'illumination global utilisé pour simuler l'interaction de la lumière avec différents objets de manière réaliste. Il est capable de simuler la réfraction de la lumière à travers une substance transparente, comme l'eau ou le verre, les inter-réflexions diffuses entre objets éclairés, et certains effets volumiques produits par des milieux comme le brouillard ou la fumée.

Radiosité : La radiosité est une technique de calcul d'éclairage (ou illumination) d'une scène 3D. Elle utilise les formules physiques de transfert radiatif de la lumière entre les différentes surfaces élémentaires composant la scène. L'illumination est dite globale car l'illumination de chaque surface élémentaire ne peut être calculée séparément des autres et le système modélisant l'ensemble des transferts ne peut être rendu que globalement.

BRDF : *Bidirectional reflectance distribution function* ou fonction de distribution de la réflectance bidirectionnelle. C'est un paramètre pondérateur de la fonction de luminance réfléchie en un point, qui dépend du matériau de la surface réfléchissante au point de réflexion, de la longueur d'onde du rayon incident, des directions incidentes et réfléchies au point de réflexion ainsi que de la géométrie de la surface.

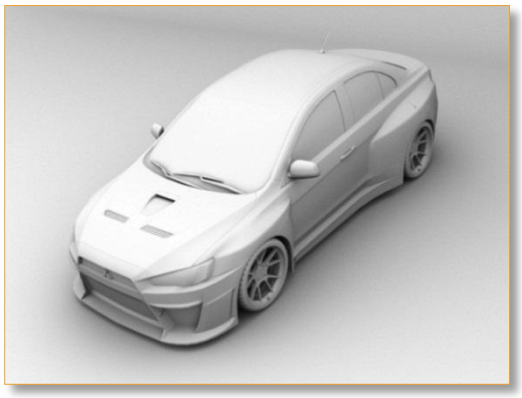
Luminance: En physique, la **luminance** d'une source est l'intensité dans une direction donnée, divisée par l'aire apparente de cette source par unité d'angle solide.

Eclairement: L'**éclairement** désigne la puissance des radiations électromagnétiques par unité de surface.

Pinhole camera / sténopé : Pour faire simple, on peut dire que le sténopé fonctionne de la même façon que l'œil, il capture des images inversées du visible.

Orthographic camera: Une projection sans point de fuite, les parallèles restent parallèles, et les objets apparaissent toujours à la même taille. Parfait pour des vues en coupe ou les vues isométriques.

L'Ambient occlusion : C'est une méthode qui calcule uniquement l'atténuation de la lumière due à l'occlusion d'un objet et ne tient pas compte de sa matière ou de la couleur.
Principe: En émettant des rayons dans toutes les directions depuis un point d'une surface, ceux qui atteignent l'arrière-plan ou une distance définie accroissent la luminosité de la surface, alors que ceux qui rencontrent une autre surface ne donnent pas d'illumination.



Rendu d'une image en ambient occlusion.

Fichier .hdr : Un fichier image .hdr est un fichier image qui est codé avec le format RGBE. Il stocke les pixels dans un octet pour chacune des composantes RGB (rouge, vert, bleu) avec un 4eme octet partagé. Il stocke donc quatre octets par pixel.
Le plus grand avantage du RGBE est qu'il permet au pixel d'avoir une plage étendue de valeur. Souvent, lorsque les images sont générées à partir des simulations de lumière, la gamme de valeurs de pixels est beaucoup plus grande que celle du format standard 0 - 255. En conséquence, les pixels clairs sont soit ramenées à 255 soit finissent par perdre toute leur précision. En utilisant un exposant commun, les gains du format RGBE nous permettent de ne pas utiliser des valeurs en virgule flottante codées sur 12 octets par pixel. Le format permet de gérer des pixels très lumineux, sans perte de précision pour les zones foncées.

Les nombres à virgule flottante : Les nombres à virgule flottante sont les nombres les plus souvent utilisés dans un ordinateur pour représenter des valeurs non entières. Ce sont des approximations de nombres réels. Les nombres à virgule flottante possèdent un signe ***s*** (dans {-1, 1}), une mantisse ***m*** (aussi appelée significande, *voir ci-dessus*) et un exposant ***e***. Un tel triplet représente un réel ***s.m.b^e*** où b est la base de représentation. En faisant varier ***e***, on fait « flotter » la virgule décimale. Généralement, ***m*** est d'une taille fixée. Ceci s'oppose à la représentation dite en virgule fixe, où l'exposant ***e*** est fixé. Un nombre à virgule flottante peut être simple précision ou double précision.

	Encodage	Signe	Exposant	Mantisse	Valeur d'un nombre	Précision	Chiffres significatifs
Simple précision	32 bits	1 bit	8 bits	23 bits	$(-1)^S \times M \times 2^{(E-127)}$	24 bits	7
Double précision	64 bits	1 bit	11 bits	52 bits	$(-1)^S \times M \times 2^{(E-1023)}$	53 bits	16

Un arbre binaire est une structure de données qui peut se représenter sous la forme d'une hiérarchie dont chaque élément est appelé nœud, le nœud initial étant appelé *racine*. Dans un arbre binaire, chaque élément possède au plus deux éléments fils au niveau inférieur, habituellement appelés *gauche* et *droit*. Du point de vue de ces éléments fils, l'élément dont ils sont issus au niveau supérieur est appelé *père*.

Remerciements:

Je tiens à remercier tout particulièrement Eric Coiro pour l'offre de stage, pour son accueil, pour sa disponibilité, ses explications, et sa gentillesse. Je remercie aussi le centre Onera de Salon de Provence qui m'a permis de passer ce stage dans les meilleures conditions mais aussi les stagiaires (Gwen, Badisse, Thomas, Jacques), doctorant (Nicolas) et personnels du centre (Nicolas) avec qui j'ai passé de très bons moments. Enfin je remercie la Patrouille de France et la base militaire 701 pour le cadre et les spectacles aériens pendant nos pauses repas.

I. Présentation

1. Le stage et son rapport

Mon stage au sein de l'Onera s'est déroulé du 19 mars au 17 aout. Durant ce stage, je devais mettre en place un découpage de l'espace pour accélérer le raytracer sous CUDA et ainsi accélérer la pass2 du logiciel Crira. La recherche d'une solution technique a été effectuée dans une première étape, et nous a conduit à choisir le développement d'un nouvel outil sous Optix.

Mon stage fait appel à des notions techniques que nous n'avons pas l'habitude de rencontrer, j'ai construit le rapport de tel sorte qu'elles soient expliquées au fur et à mesure afin d'amener le lecteur à tout comprendre sans l'aide de documents annexes. Certaines parties ne sont là que dans un but d'informations et ne sont pas issues de mon travail, elles sont alors clairement identifiées. Voici la description de la construction du rapport:

La partie I est une présentation générale de l'entreprise, du laboratoire d'accueil, du projet et de l'outil Crira.

La partie II introduit les notions les plus importantes à la compréhension de mon travail. Elle est rédigée par mes soins mais elle s'appuie sur de nombreux ouvrages et documents.

La partie III présente la problématique et le travail de recherche d'une solution technique. Elle est issue en grande partie de mes recherches et du travail des étudiants précédents.

La partie IV contient une description détaillée de l'outil graphique que j'ai développé dans le cadre du stage, avec les étapes de définitions, de programmations, les tests et le fonctionnement du logiciel mis en place. Une étude comparative de divers algorithmes de partitionnement de l'espace a été réalisée pour des géométries représentatives de celles utilisées dans Crira. La méthode la plus adaptée à notre problématique est proposée en fonction du critère du nombre de facettes. C'est le cœur du rapport.

La partie V complète la partie IV. Elle contient un descriptif des principaux avantages et inconvénients des méthodes implémentées dans ARTGpu. Le principe de chaque méthode est tout d'abord décrit, avant une présentation comparative des méthodes, obtenue suite à un ensemble de tests réalisés avec cet outil.

La partie VI contient un projet secondaire sur le modèle thermique de l'outil Crira.

Dans les ANNEXES, la partie sur l'infrarouge a été prise et modifiée d'un ancien rapport de stage. Les fiches sur CUDA, OPTIX et ARTGpu ont été rédigées par mes soins dans le but d'offrir au personnel de l'Onera un aperçu des techniques de programmations dans ces deux langages ainsi qu'une aide sur le fonctionnement du logiciel. Le résumé sur le format bdd est issu d'un document technique interne à l'Onera.

Toutes les images de rendu 3d présentes dans ce rapport ont été créées avec ARTGpu.

2. L'Onera, The French Aerospace Lab (article issu du site de l'Onera)

A l'issue de la deuxième guerre mondiale, les fondateurs de l'**Onera** étaient convaincus « qu'à condition d'aller résolument de l'avant et de faire vite, la France pouvait encore nourrir l'ambition de se constituer une aviation moderne ».

En 1946, les équipes de chercheurs, d'ingénieurs, de techniciens et d'ouvriers, qui avaient été dispersées, furent ainsi rassemblées pour construire l'**Onera**: *Office National d'Etudes et de Recherches Aéronautiques*.
Aujourd'hui l'Onera est un **EPIC**, *Établissement Public à caractère Industriel et Commercial*, et ses activités en font le premier acteur français en recherche et développement dans les domaines de l'Aéronautique, de l'Aérospatiale et de la Défense.

Ses missions : développer et d'orienter les recherches dans le domaine aérospatial. Concevoir, réaliser, mettre en œuvre les moyens nécessaires à l'exécution de ces recherches. Assurer, en liaison avec les services ou organismes chargés de la recherche scientifique et technique, la diffusion sur le plan national et international des résultats de ces recherches. Favoriser la valorisation par l'industrie aérospatiale et de faciliter éventuellement leur application en dehors du domaine aérospatial.

A ces divers titres, il est notamment chargé :

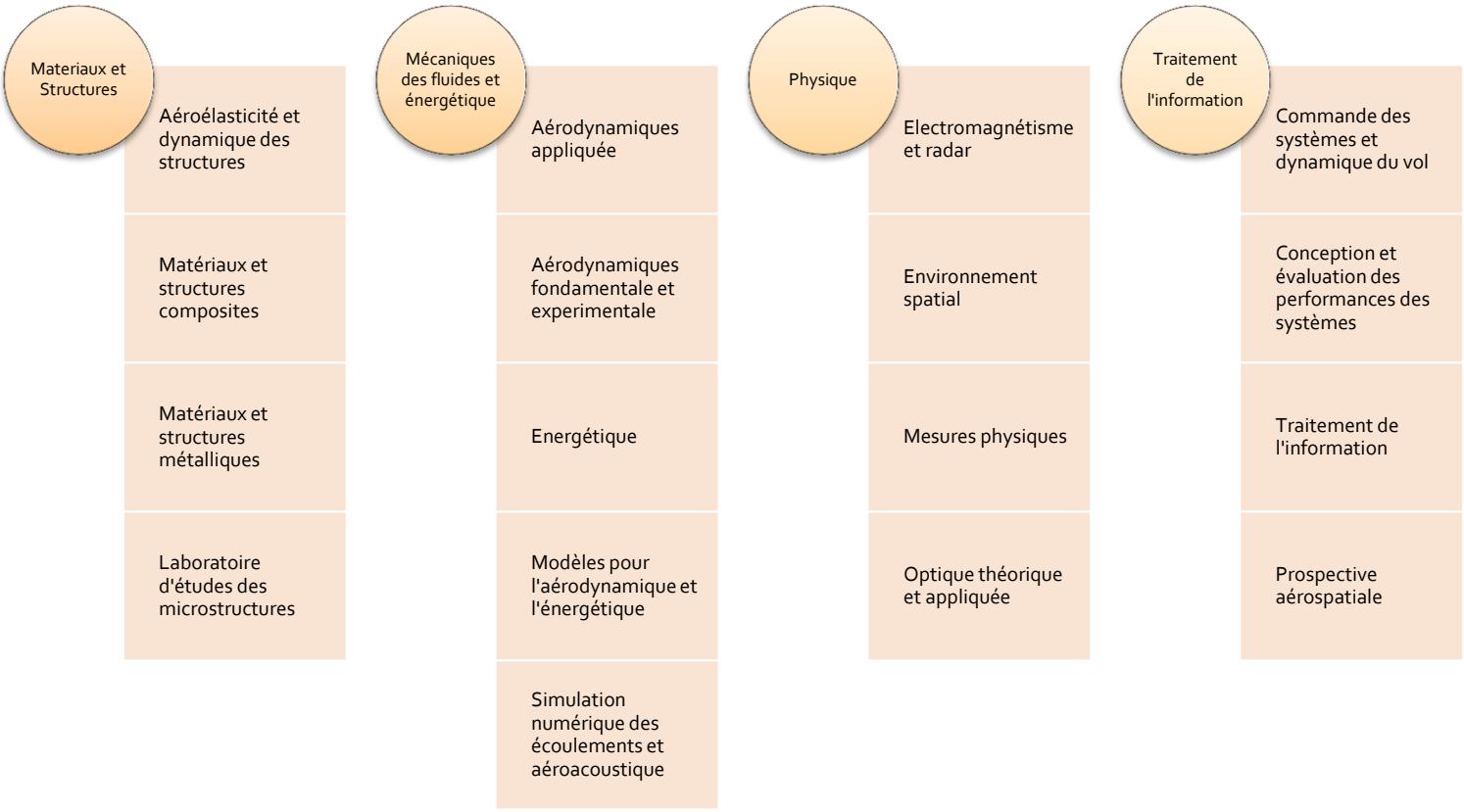
- ⇨ D'effectuer lui-même ou de faire effectuer, à son initiative ou à la demande, toutes études ou recherches intéressant l'industrie aérospatiale.
- ⇨ De réaliser des moyens d'essais et de calcul au profit de la recherche et de l'industrie aérospatiale et de les mettre en œuvre.
- ⇨ D'assurer la liaison avec les organismes français, étrangers et internationaux dont l'activité peut contribuer à l'avancement de la recherche aérospatiale.
- ⇨ D'assurer la diffusion et la valorisation des résultats obtenus, en particulier par publications, brevets, licences d'exploitation.
- ⇨ De promouvoir le lancement ou le développement d'initiatives utiles à la recherche ou à l'industrie aérospatiale.
- ⇨ D'assister, en tant qu'expert et à la demande, les organismes et services officiels.
- ⇨ D'apporter son concours, dans son domaine de compétence, à la politique de formation à la recherche et par la recherche.

Le rayonnement international de l'Onera n'a cessé de croître et 25% de ses activités sont conduits en coopération. Enfin le taux de contrats européens par chercheur est deux fois plus élevé que la moyenne des organismes français de recherche. Les compétences de l'Onera sont organisées en 17 départements (*voir graphiques page suivante*) dont un laboratoire mixte Onera/CNRS. Quatre branches scientifiques coordonnent les actions dans des domaines connexes.

L'Onera est implanté dans huit centres géographiques employant environ 2 000 personnes, dont 1 500 ingénieurs et scientifiques incluant 230 doctorants, et personnels techniques.

- ⇨ 3 centres situés en Ile-de-France.
- ⇨ 2 centres en Midi-Pyrénées.
- ⇨ 3 autres centres (Lille, Salon de Provence, Avrieux)

Le centre ONERA de Salon est hébergé sur la base aérienne 701 qui abrite également l'École de l'Air. L'organisme est sous tutelle du Chef de Cabinet du ministère de la Défense et a comme clients et partenaires, entre autres, la Direction Générale de l'Armement (DGA), EADS, Dassault Aviation et Safran.



Organisation des 17 départements de l'Onera.

3. Le DOTA

Le **DOTA** est le *Département d'Optique Théorique et Appliquée*. Il fait partie de la branche Physique de l'organisme et regroupe environ 150 chercheurs et ingénieurs, ce qui en fait l'un des plus important centre européen de recherche en optronique de défense. Son principal objectif est l'utilisation de l'optique (de l'ultraviolet à l'infrarouge lointain) et de l'optronique dans les domaines de l'aéronautique, de l'espace et des systèmes militaires, à des fins de simulations ou de détection.

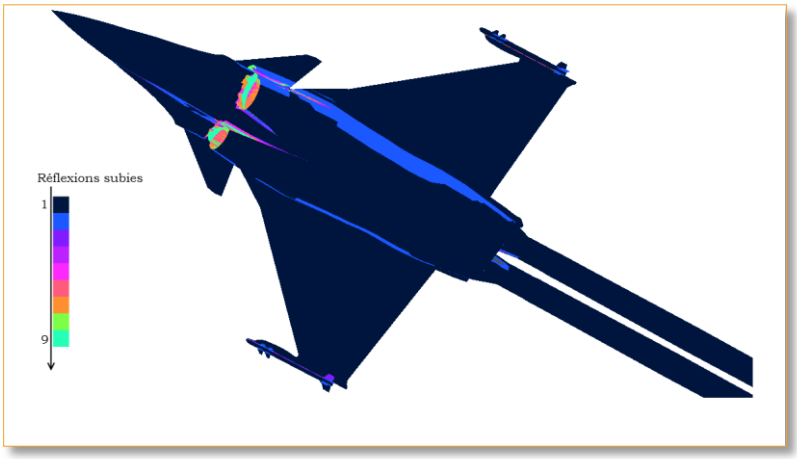
4. Crira

Le code **Crira**, *Calcul de Rayonnement Infrarouge des Aéronefs*, a pour but principal d'établir la signature infrarouge (SIR) passive d'un aéronef en vol, en fonction de paramètres d'environnement globaux et propres à l'avion et des paramètres du capteur employé pour la simulation (cette partie est bien distincte du code et est en fait un traitement sur les images calculées précédemment).
Les premières études ayant amené à la création de cet ensemble d'outils datent du milieu des années 70. Actuellement, ce logiciel utilise un **raytracer** CPU, équipé d'un **octree**, d'un calcul de **réflexions diffuses** et effectue ses calculs d'éclairement avec des **BRDF**. Cette suite logicielle est à vocation interne et externe, fournie à la DGA et à des industriels. Le code **Crira** travaille sur scène fixe, dans laquelle les objets ne se déplacent pas.

Il se divise en quatre grandes parties, ou passes:

- ⇨ Passe 1 Calcul des paramètres généraux en fonction de la scène que l'on souhaite, et calcul du(es) jet(s).
- ⇨ Passe 2 Rendu géométrique de l'avion.
- ⇨ Passe 3 Calcul radiatif de l'aéronef en luminance, avec prise en compte des BRDF.
- ⇨ Passe 4 Génération d'images en luminance à la résolution de l'entrée capteur.

Chaque passe est représentée par un ou plusieurs exécutables. L'ensemble est assez hétérogène du point de vue des codes : sont utilisés du FORTRAN, du C et du SCRIPT SHELL. L'objectif du stage est l'optimisation du rendu géométrique, c'est la passe2 qui va nous intéresser.



L'échelle de couleurs indique le nombre de réflexions sur la surface de l'avion (source : Crira, Dota).

5. Le Projet

Ce stage se déroule du 19 mars au 17 août 2012, pour l'Université Montpellier 2 et au sein de l'unité DOTA du Centre de Salon de Provence de l'Onera. L'objectif principal est l'implémentation d'un algorithme de subdivision de l'espace sous GPU au sein du code CRIRA, dans le but de rendre l'exécution de ce code et l'utilisation du raytracer GPU beaucoup plus rapide (*voir la lettre de mission ci-dessous*).

DESCRIPTION DU STAGE

Domaine d'étude : Lancer de rayons sur GPU

Type de stage ☒ Fin d'études bac+5 ☒ Master 2 recherche ☐ Bac+2 à bac+4

Intitulé : Implémentation d'une technique de subdivision de l'espace et intégration dans un code de signature d'aéronefs.

Sujet : Le Département Optique Théorique et Appliquée (DOTA) développe depuis de nombreuses années des outils de modélisation de la signature infarouge des aéronefs. Il a ainsi développé le code de calcul du rayonnement infrarouge des aéronefs CRIRA. Ce code fournit des estimations de signature par une méthode de lancer de rayons stochastique.

Un portage de ce code est en cours pour réaliser des calculs parallèles sur GPU (CUDA, C) sur Windows.

Une étude préliminaire, en cours de finalisation, permettra de sélectionner le type de méthode de subdivision de l'espace la mieux adaptée au code Crira.

L'objectif du stage est d'implémenter cette méthode et de l'intégrer dans le code de signature.

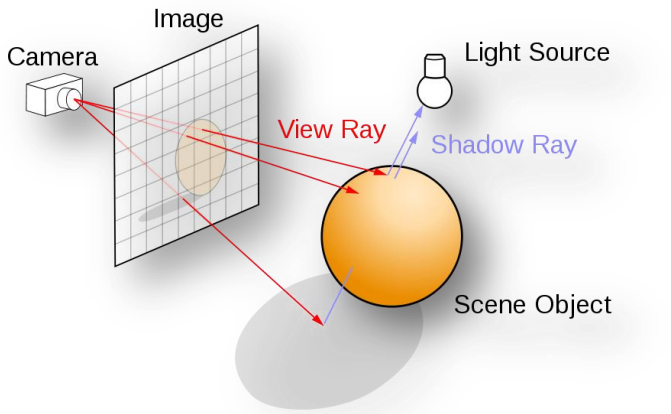
Les principales étapes du stage sont les suivantes :

- implémentation en C d'une technique de subdivision de l'espace compatible d'une utilisation avec CUDA : probablement kd-tree ou BVH,
- Intégration dans Crira
- Réalisation de cas tests représentatifs (géométries d'avions) et comparaison avec la version CPU du code

II. Introduction aux notions essentielles

1. Raytracing

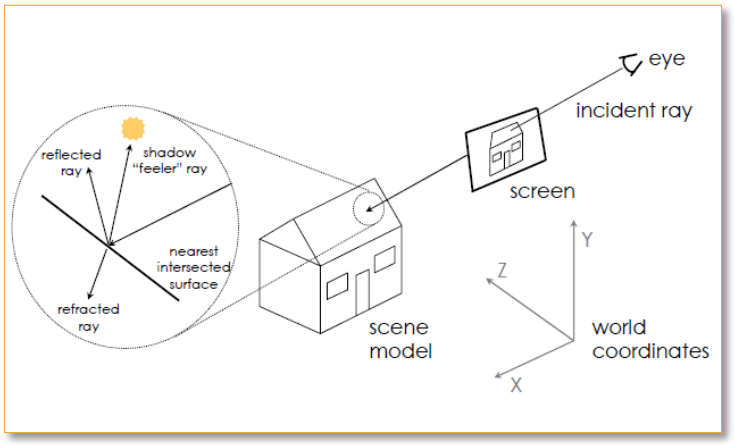
Le **lancer de rayon** est une technique de synthèse d'image qui consiste à effectuer le cheminement inverse de la lumière à partir de l'observateur vers la scène 3D, afin de déterminer la couleur de chacun des points de l'image à synthétiser. Pour lancer des rayons dans la scène 3D, on place devant l'observateur une image virtuelle (qui sera celle que l'on va synthétiser). Puis, on lance des rayons partant de l'observateur et passant par chacun des pixels de l'image virtuelle afin de déterminer leur correspondance dans la scène 3D.
Une fois le premier point d'intersection du rayon avec la scène trouvé, on calcule la couleur de ce point afin de déterminer la couleur du pixel de l'image virtuelle correspondante. Pour cela, on va relancer différents rayons dans la scène afin de déterminer quelles sources de lumière éclairent le point, quels autres objets de la scène se réfléchissent ou se réfractent en ce point. Pour les nouveaux points d'intersection avec d'autres objets de la scène ainsi trouvés, on peut réitérer le même processus plusieurs fois selon la qualité du rendu souhaitée.



Exemple de tracé de rayons dans une scène (source : raphael-brugier.fr).

Principe détaillé

On trace un rayon primaire passant par un pixel avec **P** le point d'intersection. On calcule la contribution de la source vers P en traçant les rayons d'ombres. Si un rayon d'ombre croise un objet opaque entre P et la lumière source alors P est l'ombre. On calcule la contribution à P des autres points de la scène en traçant des rayons secondaires: réfléchies et réfractées. Un rayon réfléchi est tracée seulement si le matériau est spéculaire. Un rayon réfracté est tracée seulement si le matériau est transparent. Si un rayon secondaire croise la scène à un point P', on calcule la contribution des sources à P' en traçant les rayons d'ombres. Chaque rayon apporte sa contribution à la luminance d'un point.



Exemple détaillé de raytracing (source alrg.org).

2. Path Tracing et Photon Mapping

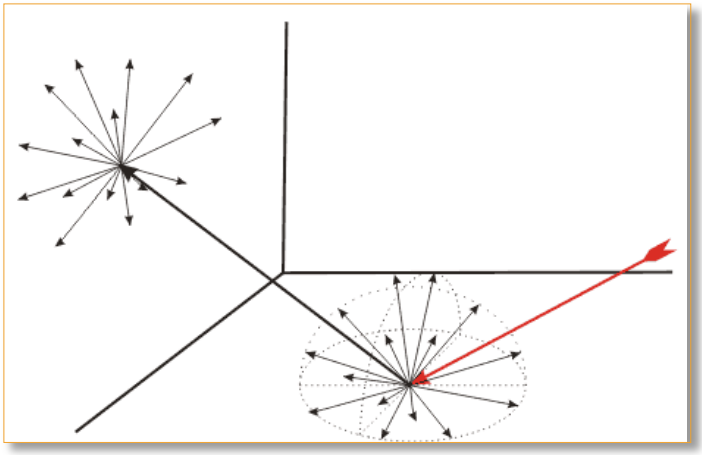
L'illumination globale est un point critique dans la génération d'images de synthèse réalistes. Les algorithmes d'illumination globale prennent en compte non seulement la lumière partant directement d'une source de lumière (*illumination directe*), mais aussi la lumière venant d'une réflexion (*illumination indirecte*). Si l'on souhaitait reproduire une simulation parfaite de ce phénomène on n'en aurait pas le temps de notre vivant car la puissance nécessaire pour calculer les chemins de chaque photon serait trop importante. Aussi, de nombreuses techniques d'approximation ont été élaborées. La technique d'Illumination Globale est la plus intéressante pour créer des images de synthèse qui tendent à être aussi réalistes que des images réelles. Ces algorithmes calculent avec approximation **l'équation de rendu** qui décrit le chemin et l'interaction de la lumière avec son environnement.

Le Path Tracing

Le **path tracing** est une technique de lancer de rayon (*ray tracing*), utilisée pour déterminer l'illumination globale d'une scène 3D, pouvant aller jusqu'à générer l'image finale par une constitution progressive. L'image n'est d'abord qu'un brouillard de pixels qui s'affine progressivement jusqu'à être débarrassée presque complètement de son « grain ».

Cette technique de rendu d'images 3D consiste à lancer des rayons dans des directions aléatoires (*exemple ci-contre*) depuis l'objectif de la caméra, jusqu'à ce que ces rayons atteignent une surface géométrique. À chaque impact, le rayon va rebondir une nouvelle fois, en perdant de l'énergie à chaque rebond, jusqu'à atteindre la couleur noire ou une source lumineuse. Les luminosité et couleur du tout premier point d'impact seront donc une pondération de la lumière et la couleur de tous les rebonds qui ont eu lieu pour un rayon.

L'intérêt de cette méthode est qu'elle n'est pas limitée dans la quantité de rebonds et réalise un modèle de la lumière ambiante intégral.



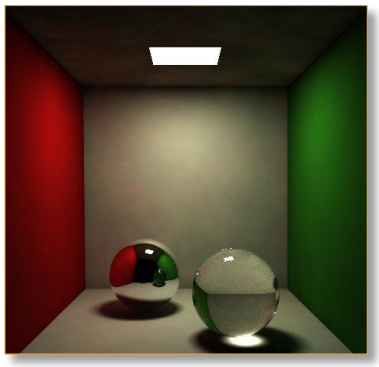
Tracé d'un rayon en path tracing
(source : http://graphics.ucsd.edu).

Parmi les différentes techniques disponibles, ARTGpu intègre la méthode du Path Tracing bidirectionnelle (intégration de l'illumination dans le path tracing) qui sera expliquée dans la partie V.

Le Photon Mapping

Le **photon mapping** est un algorithme en deux passes développé par Henrik Jensen comme une alternative au raytracing.

Il sépare les informations d'illuminations et les objets 3d et la solution est stockée dans une structure de donnée spatiale (comme les kd-tree) que l'on appel photon map. Le photon mapping permet aussi d'afficher des effets tels les Caustiques. L'image ci-contre montre une boite de Cornell rendu avec la technique du photon mapping.



Les Caustiques

Une **caustique** désigne en optique et en mathématiques l'enveloppe des rayons lumineux subissant une réflexion ou une réfraction sur une surface ou une courbe. Plus spécifiquement, on parle de caustique *au flambeau* lorsque les rayons lumineux sont issus d'un point à distance finie et de caustique *au soleil* si la source lumineuse se trouve à une distance infinie. Une caustique par réflexion est aussi appelée *catacaustique*, tandis qu'une caustique par réfraction est appelée *diacaustique*.

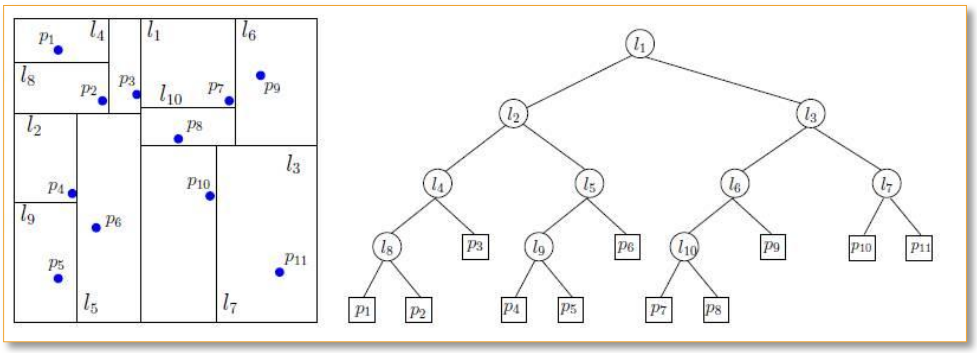
Ces phénomènes sont très difficiles voire impossibles à modéliser par les méthodes classiques de rendu, c'est pour cela que l'on utilise le photon mapping ou la path tracing pour faire apparaître ces effets.

3. Les structures accélératrices

Une structure accélératrice est en fait un type d'arbre binaire qui permet, dans le domaine du rendu graphique, d'accélérer les calculs en recherchant au préalable les intersections entre les rayons et les géométries, mais aussi en faisant du stockage comme dans le photon mapping avec la photon map. Ces arbres fonctionnent en deux étapes : leurs créations et leurs parcours. Les données à l'intérieur de l'arbre sont stockées sous la forme d'une hiérarchie dont chaque élément est appelé nœud, le nœud initial étant appelé racine. Chaque nœud peut avoir des fils et ainsi de suite. Si un nœud n'a pas de fils, il devient une feuille.
Il existe trois grandes familles de structures accélératrices : les grilles régulières, les partitionnements de l'espace adaptatif et les volumes englobants. Nous porterons notre attention sur plusieurs structures appartenant aux deux dernières familles.

Le Kd-Tree

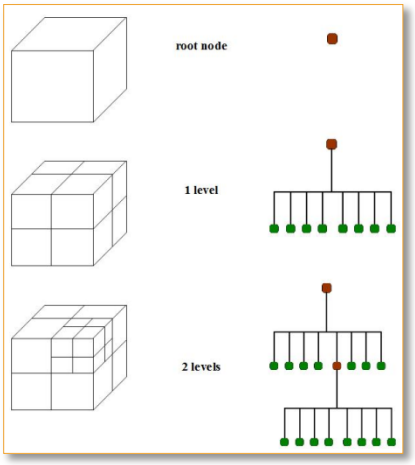
Le **kd-Tree**, *k-dimensional tree*, est un partitionnement spatial de l'espace à k-dimensions permettant de structurer les données en fonction de leur répartition dans l'espace. Il est un cas particulier des "Binary Space Partitionning (BSP) trees". Les BSP trees subdivisent l'espace à k-dimensions en coupant chaque volume englobant en deux sous volumes par un plan de l'espace, et en réitérant récursivement sur ces deux nouveaux volumes ainsi obtenus. Dans le cas du kd-Tree, ces plans séparateurs sont toujours choisis de telle façon que leur normale soit un des axes du système de coordonnées de l'espace. Cela permet de simplifier la construction, mais aussi le parcours de l'arbre. Le rôle du kd-Tree est double : il permet, d'une part, d'avoir une subdivision spatiale optimisée de l'espace permettant d'accélérer le traitement des données et, d'autre part, de stocker les données sous la forme d'un arbre binaire.



Exemple de découpage de l'espace avec un kd-tree (source google image).

L'Octree

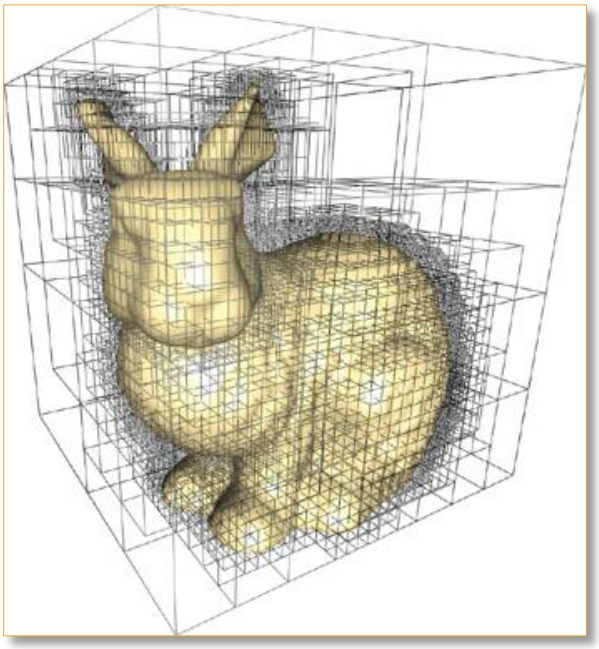
L'**Octree** est une structure de données hiérarchiques (cas particulier du kd-tree) basée sur une décomposition récursive d'une région 3D. Chaque nœud représente un cube de cette région, et le nœud racine représente la région entière. Chacun des nœuds peut avoir 8 fils. Imaginons une grosse boite qui englobe toutes nos faces, cette boite se nomme une Bounding Box ou AABB (Aligned Axis Bounding Box). Dans cette AABB, nous allons compter le nombre de faces, si ce nombre est supérieur à une certaine valeur x, alors on coupe notre boite en 8 et on recommence jusqu'à avoir moins de faces que x. Résultat, chaque nœud a soit 8 fils, soit aucun (une feuille). Dans chacun de ces nœuds nous avons une liste des faces qu'il contient.



Exemple de partitionnement de l'espace avec un Octree (source google image).

Le BVH

Un **BVH** est un arbre de volumes englobant. En bas de la hiérarchie, la taille du volume est juste assez grande pour englober un objet unique (ou dans certains cas une petite fraction d'un objet). Quand on monte dans la hiérarchie, chaque nœud a son propre volume qui englobe bien tous les volumes du dessous.
A la racine de l'arbre, on trouve un volume qui englobe tous les volumes de l'arbre et donc de la scène.

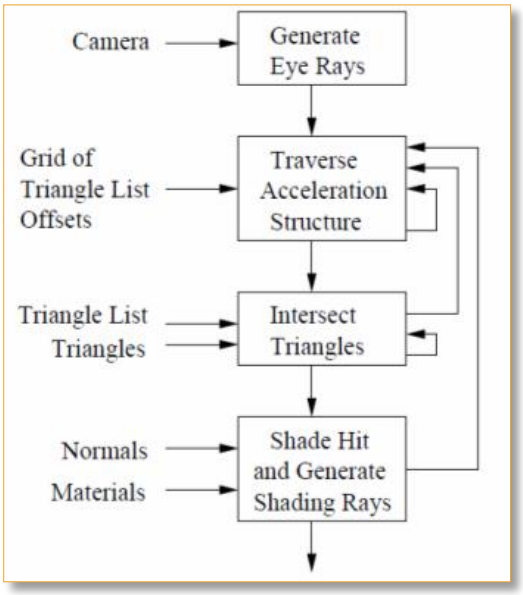


Exemple de partitionnement de l'espace autour d'un objet par la méthode du kd-tree (source google image).

Pour accélérer la création et le parcours de ces structures de données (type kd-tree ou bvh, l'octree devient beaucoup plus rare) on utilise des algorithmes spécifiques. La **SAH** (*surface area heuristic)* permet d'estimer à priori le coût de parcours de l'arbre lors de sa construction. Cela permet de choisir, pendant la construction, les plans séparateurs et les critères de fin qui minimiseront ce coût.

4. Les structures accélératrices et le raytracing

Une des tâches primordiales du lancer de rayons est donc de calculer les points d'intersection entre les rayons et la scène 3D. Un kd-Tree (ou une autre structure accélératrice) va permettre d'accélérer ce traitement entre rayons et géométries qui est l'étape la plus couteuse en temps de calcul. En effet, au lieu de tester les intersections entre un rayon et tous les objets (ou triangles) de la scène, on teste les intersections uniquement avec les objets contenus dans les volumes traversés par le rayon. Il va donc falloir parcourir le kd-Tree de volume en volume en testant s'il y a intersection ou non avec les objets du volume traversé s'il n'est pas vide. Plus le rayon traverse de volumes vides, moins il y a d'intersections avec des objets à calculer, ce qui permet de diminuer le temps nécessaire pour trouver le point d'intersection du rayon avec la scène. C'est pourquoi il était intéressant de maximiser la taille des volumes vides lors de la construction du kd-Tree.



Déroulement d'un raytracing avec structures accélératrices (source Nvidia).

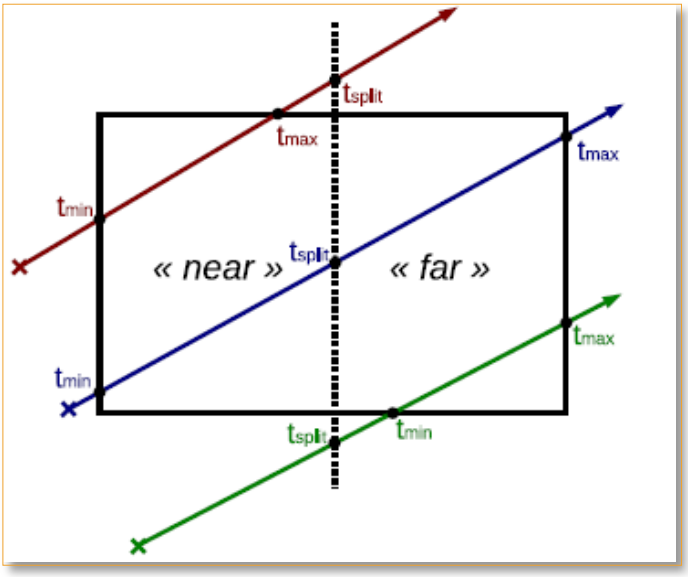
Un algorithme de parcours d'un kd-Tree dans le cas du lancer de rayon prend en entrée le kd-Tree et le rayon, et rend en sortie le premier point d'intersection du rayon avec la scène. Cet algorithme récursif parcourt exactement une fois tous les nœuds et toutes les feuilles dont le volume correspondant est traversé par le rayon. Quand le rayon entre dans un nœud du kd-Tree qui a deux fils, l'algorithme décide si les deux fils sont traversés et dans quel ordre. Il détermine, parmi les deux fils du nœud traversé, le nœud le plus porche (*near*) et le plus éloigné (*far*), en fonction de la position de l'origine du rayon par rapport au plan séparateur. Lorsque le rayon traverse uniquement l'un des deux fils, l'algorithme descend dans ce nouveau nœud et réitère. Lorsque le rayon traverse les deux fils, l'algorithme sauvegarde les informations du fils *far*, descend dans le fils *near* et réitère. Si aucune intersection n'est trouvée à l'intérieur de ce fils *near*, l'algorithme va alors dans le fils *far* et réitère.

Pour l'implémentation, on utilise les valeurs tmin et tmax qui correspondent aux distances d'entrée et de sortie du rayon dans le nœud visité par rapport à l'origine du rayon (*voir figure ci-dessous*). On calcule également, à chaque itération, la distance tsplit qui correspond à la distance entre le point d'intersection du rayon avec le plan séparateur et l'origine du rayon.

Ces trois valeurs permettent alors de déterminer quels fils sont traversés par le rayon et dans quel ordre :

- ⇨ si tsplit > tmax, alors le rayon traverse seulement le fils *near.*
- ⇨ si tsplit < tmin, alors le rayon traverse seulement le fils *far*.
- ⇨ si tmin < tsplit < tmax, alors le rayon traverse en premier le fils *near*, puis le fils *far*.

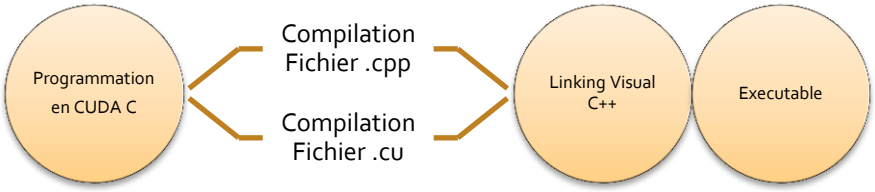
Enfin, la meilleure implémentation utilise une pile pour stocker le fils *far* dans le cas où les deux fils doivent être visités, afin d'éviter la récursivité.



Démonstration du choix du fils à traverser dans une structure accélératrice (source Cedric Fleury).

5. La programmation sous GPU : Cuda

CUDA est un environnement logiciel qui permet d'utiliser le GPU au travers de programme de haut niveau comme le C ou le C++. Le programmeur écrit un programme en C avec des extensions CUDA, de la même manière qu'un programme en OpenMP par exemple. CUDA nécessite une carte graphique équipée d'un processeur Nvidia. Pour notre logiciel ARTGpu, il est recommandé que la carte soit de type Fermi ou Kepler car les calculs pour le path tracing nécéssitent des variables à double précisions. Les fichiers sources doivent être compilés avec le compilateur nvcc fourni avec la suite CUDA.

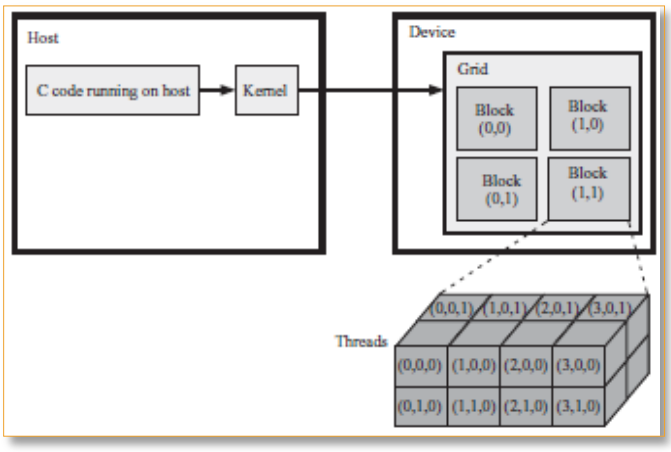


Déroulement de compilation pour un programme CUDA / C.

Un programme CUDA utilise des **kernels et des flux de données**. Ces flux de données peut être par exemple, des vecteurs de nombres flottants, ou des ensembles de frames pour du traitement vidéo. CUDA fournit trois mécanismes pour paralléliser un programme : un regroupement hiérarchique des **threads**, des mémoires partagées, des barrières de synchronisation.

Répartition du travail entre threads regroupés en bloc

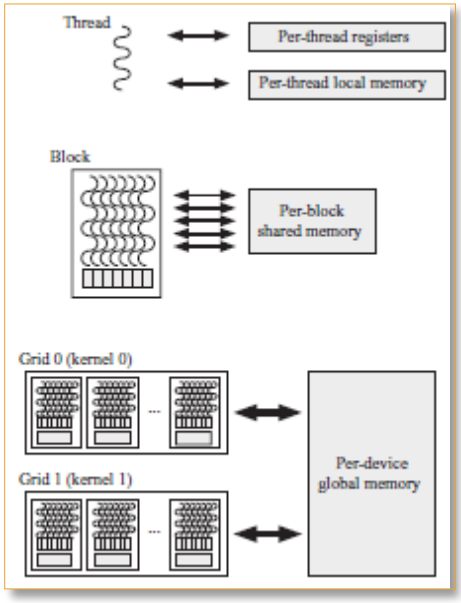
Chaque **kernel** est associé avec une grille dans le **device**. Le choix du nombre de **threads** et de **blocs** est conditionné par la nature de l'application et la nature des données à traiter. Le but est d'offrir au programmeur des moyens d'organiser les threads de manière adaptée à l'organisation des données, afin de simplifier l'accès à ces données : les données sont organisées en **grille** et les threads aussi.
Chaque thread dispose d'un identifiant unique, ID, appelé **threadIdx**. Les variables **blockDim** et **threadIdx** sont définies automatiquement et sont de type dim3.
Les threads dont un bloc peuvent être organisés suivant une, deux ou trois dimensions. Chaque dimension est accédée par la notation **threadIdx.x**, **threadIdx.y** et **threadIdx.z**.



Exécution d'un kernel sur un device (source libpfb.so).

La mémoire

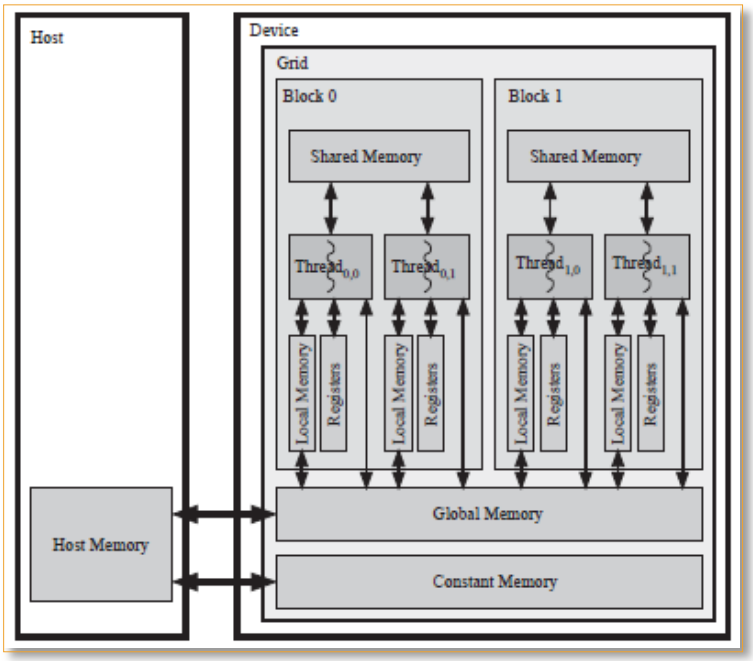
La gestion de la mémoire sur le GPU est une chose importante pour les programmes CUDA car elle aborde les problèmes d'accessibilité en lecture/écriture de la mémoire et de vitesse d'exécution des programmes.
Les différents types de mémoire présents sur une carte graphique sont : les registres, la mémoire partagée, la mémoire locale, la mémoire constante, la mémoire texture et la mémoire globale. Elles se caractérisent par leurs capacités de stockage, de vitesse et de facilité d'accès par le GPU et par leur accessibilité en lecture/et ou écriture par le CPU.



Allocution des différentes mémoires (source libpfb.so).

La communication Host - Device

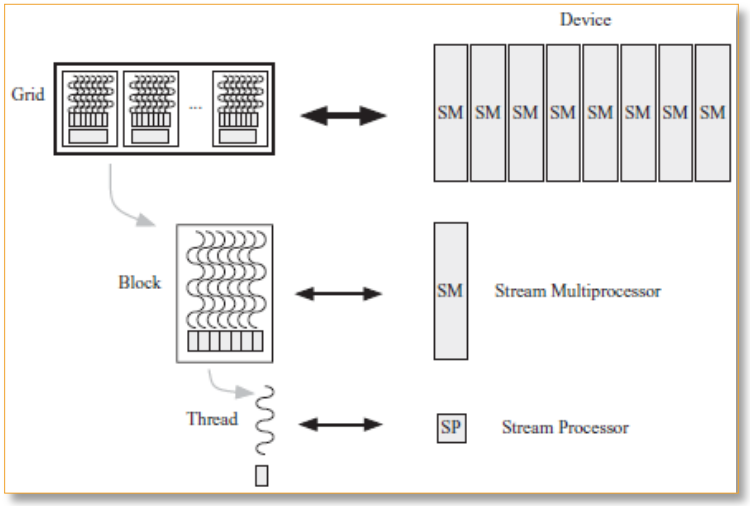
La mémoire globale en bas du schéma est le moyen de communiquer des données entre l'**hôte** et le **device**. Le contenu de la mémoire globale est visible depuis toutes les threads et est accessible en lecture / écriture, celle indiquée constant memory, n'est accessible qu'en lecture seulement. La mémoire partagée par bloc est visible depuis tous les threads de ce bloc. La mémoire locale, comme les registres, n'est visible que du thread.



Communication entre le CPU et le GPU (source libpfb.so).

Exécution d'une application en parallèle sur le device

Le kernel est exécuté sur une grille de blocs de threads (*voir image ci-dessous*). Différents kernels peuvent être exécutés par le device à différents moments de la vie du programme. Chaque bloc de threads est exécuté sur un multiprocesseur à flux, streaming multiprocessor : *SM*. Le SM exécute plusieurs blocs de threads à la fois.
Des copies du kernel sont exécutées sur le streaming processor : SP ou thread processors, qui exécute un thread qui évalue la fonction. Chaque thread est alloué à un SP.



Hiérarchie du GPU (source libpfb.so).

6. Optix, Raytracing Engine

Les GPU sont les outils les plus puissants lorsque l'on veut mettre en place une solution avec un haut degré de parallélisme. Et le raytracing en requiert un très grand. Cependant, ces algorithmes de raytracing sont très irréguliers, ce qui est un problème lorsque l'on veut utiliser toute la puissance de calcul d'un GPU.

OptiX est un cadre de traçage de rayons programmable permettant aux développeurs de logiciels de construire rapidement des applications de traçage de rayons obtenant des résultats extrêmement rapides à travers les GPU NVIDIA avec programmation C conventionnelle. Au lieu d'un outil de rendu avec un aspect prédéfini, ou d'être limité à certaines structures de données ou à un langage limité au rendu, le moteur OptiX est extrêmement général, ce qui permet aux développeurs d'accélérer toute tâche souhaitée de traçage de rayons et de l'exécuter sur une large palette de matériel disponible, le tout sans licence.
Optix est composé de deux parties:

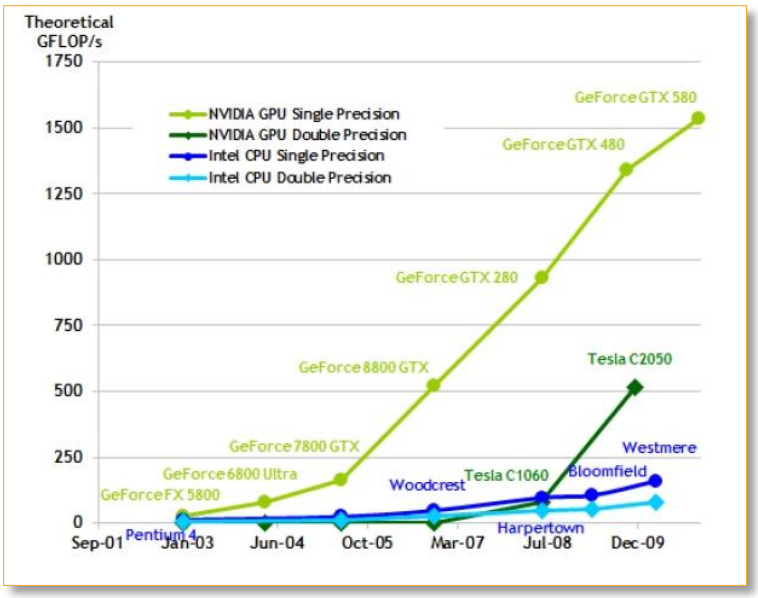
- Un API (***application programming interface)*** qui définit la structure d'un raytracer.
- Un système de programmation en CUDA C qui produit les nouveaux rayons, les intersections avec les surfaces, les réponses à ces intersections… tout cela en parallèle.

III. Problématiques et solutions

1. La situation actuelle

Nous avons vu dans la partie précédente que le logiciel **CRIRA** a été développé dans les années 1970. Bien qu'aillant eu de nombreuses améliorations et modifications, l'implémentation native en C du lancer de rayons sans parallélisation montre ses limites. A une époque où les modèles géométriques sont de plus en plus détaillés, il est urgent de trouver une solution afin d'accélérer ce processus.

Plusieurs étudiants sont passés avant moi. Charles Thierry, étudiant en stage à l'Onera en 2010 a mis en place un raytracer sous Cuda sans structures accélératrices. Bien que cette solution soit plus rapide qu'un CPU classique, le raytracer mettait plus de temps que le couple CPU / Octree de Crira. Il fallait donc absolument intégrer une structure d'arbre dans le raytracer. Ce fut le rôle de l'étudiant de 2011, qui a fait une étude sur les différents types de structures et leurs performances respectives. Mais de nos jours, les technologies et techniques évoluent tellement vite, comme le montre le graphique ci-dessous, qu'il faut trouver une solution de développement évolutive, intuitive et qui ne sera pas dépassée dans les années à venir.



Evolution des performances des GPU et CPU (source google image).

2. Les possibilités

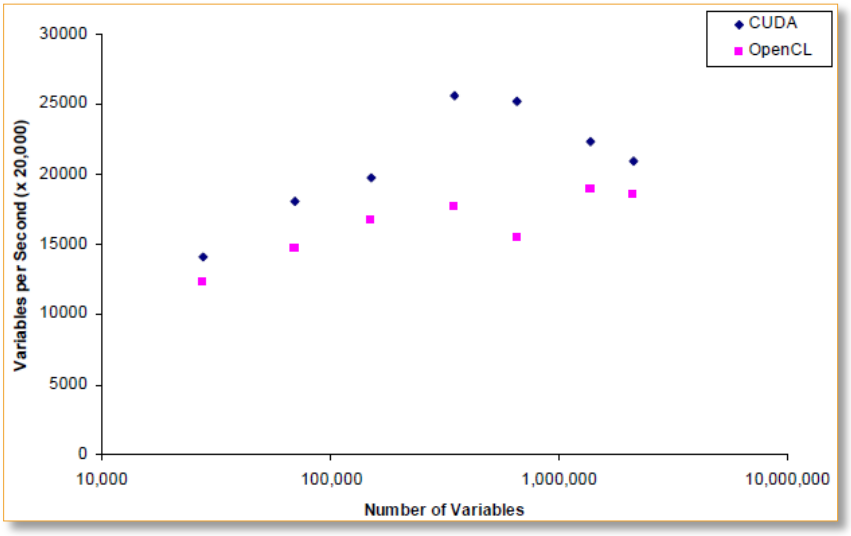
J'ai réalisé dans un premier temps une recherche bibliographique afin d'identifier des librairies permettant de générer des structures accélératrices. Il s'avère que la programmation d'une telle structure sous GPU est complexe et sort du champ de ce stage (un algorithme récursif de CPU ne pouvant pas être implémenté sur un GPU). L'idée retenue est soit d'utiliser une structure et l'adapter à la structure de données de Crira, soit d'utiliser une librairie plus générique, ne nécessitant pas d'adaptation de la structure de données. La première possibilité n'est pas évolutive, et de plus, le choix d'une méthode n'est pas optimal selon le contexte d'emploi.

Si on parle maintenant de paralléliser du code source sous GPU, deux choix s'offrent à nous: **CUDA** avec sa maturité, sa grande communauté, et les nombreux cours et aides présentes ou bien OpenCl, plus généraliste, qui permet de lancer le programme sur n'importe quelle machine.

Voici les différentes possibilités offertes aujourd'hui. Mais il faut savoir que ce domaine est en pleine expansion et est très à la mode en ce moment car très jeune.

3. Les choix

Bien qu'OpenCl offre une interopérabilité intéressante et une facilité de programmation, nous décidons d'utiliser CUDA qui est un environnement de développement beaucoup plus mature et qui permet une programmation de plus bas niveau avec son langage assembleur PTX.



Comparatif de performances entre OpenCl et CUDA :
Nombres de variables traités par seconde dans un calcul (source http://arxiv.org).

Nous utiliserons aussi la librairie **Optix** qui rend l'assemblage d'un raytracer, la gestion de traversée d'une scène, les effets et la gestion de mémoire beaucoup plus simple que pour une solution programmée de A à Z. De plus, une fois l'application fonctionnelle, elle tirera partie des améliorations matérielles futures sans nécessité de modification dans le code source. Dès que cela est possible, **Optix** évite de faire appel à des comportements classiques de raytracing pour faire appel à la place à des mécanismes en CUDA C pour par exemple la gestion des ombres (*recursive rays*), les modèles de caméra…
Optix peut donc être utilisé pour faire du raytracing, du path tracing, de la détection de collision, du photon mapping… Et il peut aussi communiquer avec OpenGL ou DIrectX.

De plus dans cette librairie, les structures accélératrices sont déjà implémentées et optimisées pour fonctionner sur GPU. Voici les structures disponibles:

- ⇨ **Sbvh:** de type BVH spécialisé dans la création d'une hiérarchie pour des scènes avec des géométries statiques.
- ⇨ **BVH:** c'est l'algorithme de base pour la création d'une hiérarchie de volume. Il est performant pour la création de hiérarchie de groupes d'objets.
- ⇨ **MedianBvh:** Toujours de type BVH, il est utilisé pour les scènes dynamiques.
- ⇨ **Lbvh:** Toujours de type BVH, il est utilisé pour les scènes d'animations.
- ⇨ **TriangleKdTree:** De type KD-Tree, il est spécialisé dans les scènes comportant des géométries composées de triangles.

Nous utiliserons par défaut la structure **Sbvh** qui est la plus rapide (*voir partie IV. b . Les performances*) et la mieux adaptée au rendu de géométries que nous voulons faire.

IV. Développement et programmation

Comme nous l'avons vu dans les pages précédentes, mon sujet de stage consiste à améliorer et optimiser la pass2 du logiciel Crira de l'Onera. Nous avons fait le choix d'utiliser la librairie Optix qui correspond le plus à nos attentes et aux besoins futurs. Afin de fournir les outils nécessaires à Crira, il a été décidé de développer un logiciel de démonstration, ***ARTGpu***, qui va être une vitrine des possibilités et performances fournit par la librairie choisie ainsi qu'une base de développement. Ce logiciel permettra de faire le rendu d'objet 3d par différentes techniques (path tracing, photon mapping, raytracing) et offrira de nombreuses options (énumérées dans la suite) qui feront de ce logiciel un outil généraliste qui pourra être utilisé éventuellement dans d'autres domaines au sein de l'Onera. Ce ne sera qu'une fois ce logiciel fini que l'intégration des fonctions sélectionnées sera possible dans Crira.

1. Cahier des charges

Mais afin de bien comprendre comment fonctionne le logiciel de l'Onera, voici les grandes étapes qui constituent la pass2.

Lecture des fichiers
• Lire la base de données existantes avions
• Lire le fichier de découpage du jet
• Lire le fichier raccord jet avion
• Lire le nombre de jets pour l'avion
Ecriture des fichiers
• Ecriture du jet au format SDM
• Ecriture du fichier raccord au format ASCII
Préparation de la scène
• Ajout du jet à la bdd Avion
• Lire les coordonnées du point de visée et de la résolution de l'image
• Lire les coordonées de l'observateur
• Préparer le scénario du rendu géometrique
Calcul
• Calcul de la boite englobante
• Calcul du rendu par une méthode d'illumination globale (méthode Monte Carlo ou technique du Z-Buffer)
• Calcul des surfaces apparentes
Sauvegarde
• Sauvegarder les images brutes
• Sauvegarde des chemins des rayons de reflexions

Une fois notre logiciel de démonstration ***ARTGpu*** fini, il faudra que les fonctions intégrées dans le logiciel Crira respectent le cahier des charges suivant :

Préparation du calcul

- Lecture d'un fichier de géometries d'avion et des jets
- Convertir et assembler l'avion et le jet
- Lecture des coordonnées de l'observateur
- Récupérer les paramètres pour le calcul
- Lecture des différents matériaux

Calcul et affichage

- Faire le rendu des géometries avec mise en place d'une structure accélératrice en complément de la méthode de rendu
- Permettre le choix entre plusieurs modèles de rendu (phong, fresnel,...)
- Proposer de nombreuses options (anti aliasing, type de camera,...)
- Permettre l'intégration futures d'autres méthodes de rendu

Fichier de sortie

- Récupérer les informations sur la AABB
- Sauvegarder les fichiers de rendu
- Sauvegarder les chemins des rayons
- Sauvegarder les paramètres des géometries

Je ferais donc en sorte de préparer au mieux et de tester tous ces besoins dans ***ARTGpu*** afin de faciliter le travail d'intégration qui arrivera ensuite.

2. Découpage des projets

Afin de planifier les différentes étapes du projet, j'ai mis à jour un diagramme de Gantt. Je vous invite à vous rendre dans l'annexe où se trouve ce diagramme avec toutes les étapes détaillées.

3. Développement du logiciel Advanced Rendering Tools Gpu (ARTGpu)

Pour commencer je vais présenter mon poste de travail : c'est une station professionnelle de la marque CARRI composée d'un ***Intel i7 Extrême (8 cœurs) avec une carte graphique Nvidia Tesla C2070 et 16 Go de DDR3***. Une machine que l'on peut donc qualifier d'ultra performante.

a. Mise en place du Raytracing

Comme nous l'avons vu auparavant, les étudiants précédents ont montré le chemin et ont donné des pistes pour la conception de ce raytracing sous GPU. Cependant le choix d'utiliser Optix impliquait de tout recommencer et de tout repenser de zéro. En effet, Optix utilise des fonctions, méthodes et styles de programmation qui lui sont propre. Un code source développé sous CUDA serait très compliqué et beaucoup plus fastidieux à convertir que le réécrire en Optix / CUDA C. Mais cette étape est indispensable si on veut pouvoir avoir un outil évolutif et compétitif.
Optix étant une technologie très jeune, il n'existe pas de cours ou de ressources pour aider dans le développement. J'ai donc utilisé comme support des samples fournis avec le SDK pour me faire une base et développer ensuite ma propre solution de raytracing. Ces samples sont Open Source et peuvent être utilisés à des fins commerciales.

Je suis donc parti d'un exemple très simple de raytracing et la première étape a été de le "nettoyer" des fonctions inutiles, de le commenter et de le rendre viable à notre projet (le préparer au futur ajout). Ceci étant fait, j'ai pu mettre en place la version alpha de mon raytracer. Cette version alpha était très basique, elle chargeait un fichier de géométrie 3d obj unique et affichait cet objet dans une fenêtre dans laquelle il était possible de naviguer.

La navigation dans la scène se fait grâce à OpenGl qui se charge de l'affichage et de la fenêtre windows. Le calcul pur est réalisé par Optix et Cuda.

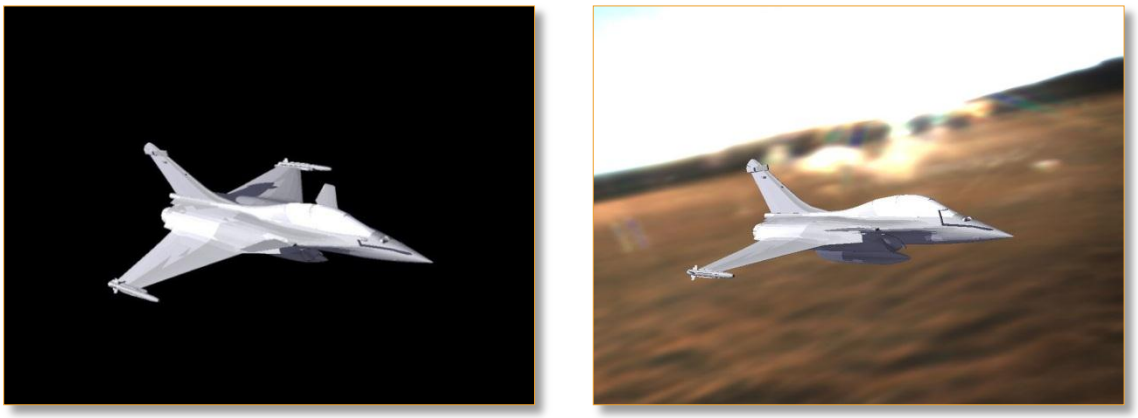
Le rendu tel qu'il était dans la version alpha du raytracing.

Chargement d'un objet différent dans le raytracer.

Le deuxième point important à régler était la possibilité de charger n'importe qu'elle fichier .obj et préparer le logiciel ***ARTGpu*** à recevoir sa futur interface. Cette Interface, qui sera un script perl, fera l'intermédiaire entre le module de rendu ***ARTRaytracing.exe*** que je développe et l'utilisateur.

Maintenant que l'on peut charger toutes les géométries possibles (seulement en fichier .obj), j'ai créé le script perl ***Interface.pl*** qui a donc pour mission de me demander le chemin du fichier géométrie, le type d'arrière-plan que l'on veut (**color** que l'on appelle aussi **miss** ou **environnement map**), le type de rendu que l'on veut (ambient occlusion, phong, fresnel reflexion, normal shader,...) et enfin si on veut relancer un calcul ou fermer l'application (on demande aussi au cas où l'on refait un calcul si on garde le même fichier .obj et le même fichier pour l'environnement map). Ce script est une interface console qui permet d'interagir avec l'utilisateur du raytracer. Toutes les informations saisies et choisies par l'utilisateur sont transmises à ***ARTRaytracing.exe*** comme arguments et sont récupérer dans l'exécutable grâce à ***argc*** et ***argv***.

Maintenant que l'interface et que la saisie des options sont en place, il a donc fallut intégrer dans le raytracer les options d'arrière-plan: ***environnement map*** et ***color*** (on choisit la couleur de fond). L'environnement map est une image .hdr qui est « plaquée sur une sphère » englobant la scène. Le fonctionnement des environnements dans un raytracing (mais aussi en photon mapping et path tracing) est le suivant : si le rayon envoyé par le raytracer ne rencontre aucune facette de la géométrie ou si il est réfléchie dans le vide alors ce rayon récupère l'information de couleur de l'image correspondante ou la couleur choisie (selon l'option choisie par l'utilisateur). C'est ce que l'on appelle une fonction ***miss***.

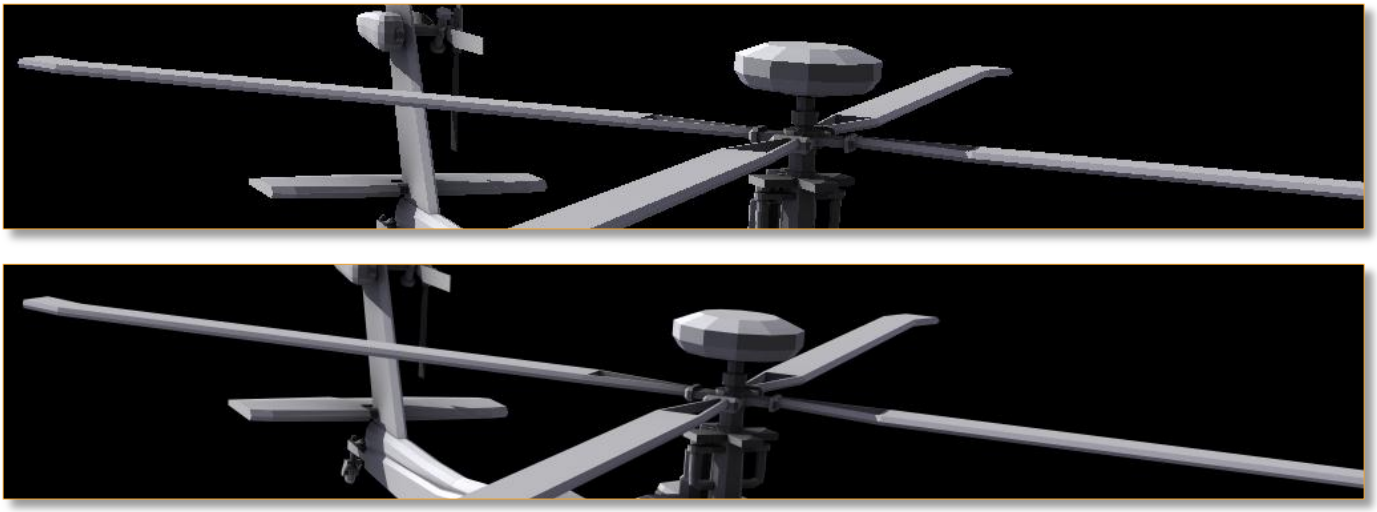


Rendu d'un objet 3d avec à gauche l'option color (miss) et à droite l'option environnement map.

Il était aussi nécessaire d'ajouter des méthodes de rendu type phong, ambient occlusion, Fresnel réflexion, normal shader. Chacun de ces modèles de rendu ont leurs spécificités. Par exemple le modèle de phong et un modèle générique qui va permettre de prendre en compte les ombres et les réflexions, le modèle de Fresnel est un modèle de phong qui va rendre les réflexions beaucoup plus réalistes car différentes selon l'angle de vue (des exemples sont visibles dans la *partie V*).

Mais pour améliorer encore le rendu, il a fallu implémenter une méthode de type antialiasing. Une image numérique est composée de pixels et selon le type d'objet que l'on a, on peut remarquer que les bords des formes ayant un angle particulier prennent la forme d'escalier : c'est le crénelage, ou aliasing. Pour supprimer cet effet visuel disgracieux, on utilise l'anticrénelage, ou *anti-aliasing*. Il y a de nombreux types d'algorithmes différents. Certains vont calculer des dégradés de couleurs autour de la région souhaitée pour adoucir le rendu d'autre vont recalculer l'image en plus grande et ensuite la réduire. Mais dans ARTGpu, pour éviter l'effet d'escalier, on renvoie un second rayon dans le pixel à une position différente du premier rayon. Ce processus va « subdiviser » le pixel et donc augmenter ces détails ce qui aura pour effet de supprimer l'effet d'aliasing.

NB : Il faut bien faire la différence entre un pixel de l'image et un pixel matériel qui compose l'écran : un pixel de l'image peut être affiché par plusieurs pixels matériels.



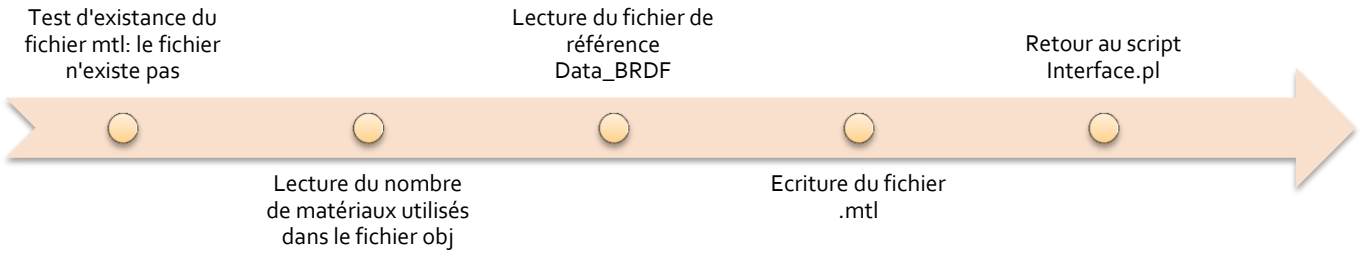
Rendu d'un objet 3d avec en haut l'option antialiasing désactivé et en bas l'option activée.

Enfin, pour rendre le rendu des objets encore plus réaliste, nous allons prendre en compte la notion de matériaux. Un objet obj pourra donc avoir une couleur unie, gérer par les coefficients de ***diffusion ambiante***, ***réflexion diffuse***, ***réflexion spéculaire*** qui sont définis dans le fichier .mtl. Si un objet est découpé en groupes, chaque groupe pourra avoir son propre matériau (tous les matériaux sont définis dans le fichier mtl, *pour plus d'informations sur ces fichiers rendez-vous dans l'annexe*).
Nous pouvons aussi assigner à des objets, ou des groupes dans un objet, des textures images qui seront appliquées sur l'objet. Le fonctionnement détaillé pour la gestion des matériaux est expliqué juste après.

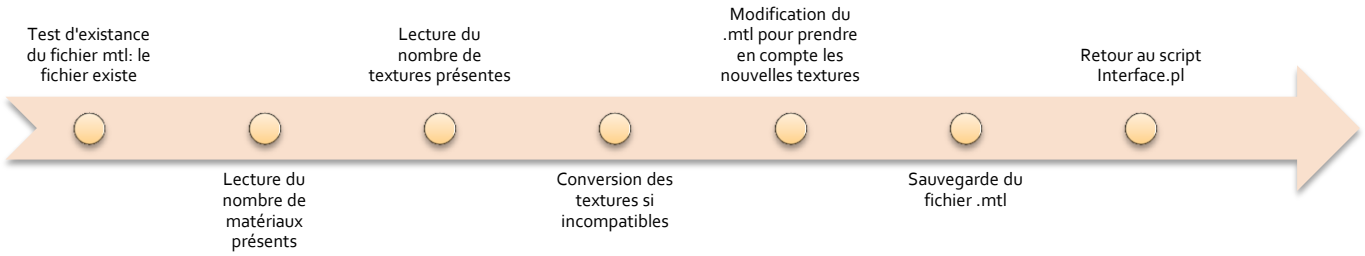
Rendu d'un objet avec à gauche une couleur unie en guise de matériau et à droite utilisation des textures images.

La gestion des matériaux

La gestion des matériaux dans ***ARTGpu*** se fait de la manière suivante, avant l'exécution des modules de rendu, le script ***Interface.pl*** fait appel au script ***matériaux.pl***. C'est ce script qui va gérer les fichiers mtl qui seront ensuite lus et chargés dans les modules. Le script ***materiaux.pl*** permet de créer automatiquement des fichiers .mtl à partir d'un fichier référence, fourni par le logiciel Crira, pour les fichiers .obj n'en ayant pas et de modifier les fichiers .mtl existants pour les rendre compatibles. Cette compatibilité passe par la conversion des textures livrées avec le .mtl qui sont souvent en .tga ou .jpg alors que notre raytracer ne supporte que les .ppm.
Voici donc étape par étape ce que fait le script de gestion des matériaux :


Test d'existance du fichier mtl: le fichier n'existe pas
Lecture du fichier de référence Data_BRDF
Retour au script Interface.pl
Lecture du nombre de matériaux utilisés dans le fichier obj
Ecriture du fichier .mtl

Génération du fichier matériau à partir du fichier de référence Data_BRDF.


Test d'existance du fichier mtl: le fichier existe
Lecture du nombre de textures présentes
Modification du .mtl pour prendre en compte les nouvelles textures
Retour au script Interface.pl
Lecture du nombre de matériaux présents
Conversion des textures si incompatibles
Sauvegarde du fichier .mtl

Modification du fichier matériau dans le cas où le fichier mtl contient des textures non compatibles.

L'étape de conversion est faite par le logiciel ***nconvert.exe*** qui est un convertisseur d'images en ligne de commande open source que j'ai intégré dans mon script perl. On fait appel à cet exécutable à chaque fois que l'on rencontre un fichier de texture dans un mauvais format.

Petit aspect technique

Pour Optix, un matériau présent dans le fichier .mtl est soit une couleur unie (gérer par les coefficients ***Ka***, ***Kd***, ***Ks***) soit une texture 2D créée à partir d'une image. Lorsque le rayon est envoyé sur une facette, il va récupérer l'information de couleur correspondante en la calculant à partir des coefficients ***Ka***, ***Kd*** et ***Ks***. Ces coefficients sont soit définis dans le fichier mtl soit récupérés sur les textures images. Il y a trois types de textures, les ***map_Ka*** qui correspondent aux textures ambiantes (et donc au Ka), les ***map_Kd*** qui correspondent aux textures diffuses (et donc au Kd) et les ***map_Ks*** qui correspondent aux textures spéculaires (et donc au Ks). Dans le cas des textures, la facette est associée à une partie de l'image, Optix va calculer la couleur à afficher en récupérant sur les images les informations RGB.
Pour résumer, si on a un matériau avec des textures, chaque facette va avoir des coefficients ***Ka***, ***Kd*** et ***Ks*** différents, ces coefficients sont définis à partir des images alors que pour un matériau avec des couleurs unies, les ***Ka***, ***Ks*** et ***Kd*** sont fixés et vont être les mêmes pour toutes les facettes.

Comparatif

Pour mieux visualiser comment interagissent les trois types de texture entre elles, j'ai effectué un petit comparatif avec le même objet. Les coefficients ***Ka***, ***Kd*** et ***Ks*** sont mis à Ø donc seules les textures ***map_Ka***, ***map_Kd*** et ***map_Ks*** vont compter. L'objet va recevoir toutes les combinaisons de texture possible afin de voir les différences et les apports de chaque partie.

Texture Ka Ks
Texture Ka Kd
Texture Kd Ks
Texture Ka Kd Ks
Texture Ka Kd Ks Ns 0.0
Texture Ka Kd Ks Ns 25.0
Texture Ka Kd Ks Ns 50.0
Texture Ka Kd Ks Ns 100.0

Comparatif de rendu selon les textures appliquées pour un même objet.

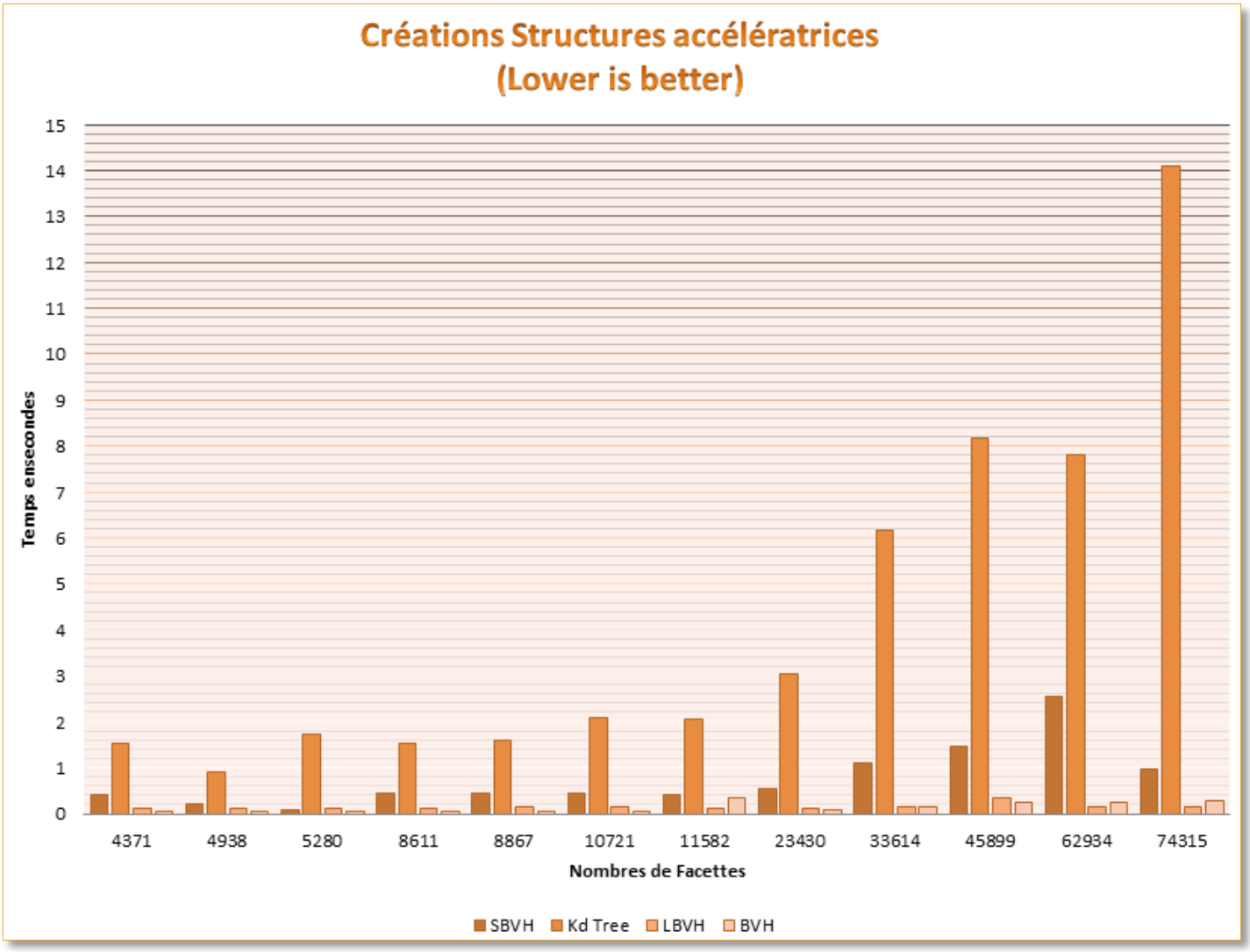
Le paramètre ***Ns*** permet de pondérer la texture spéculaire (ou le coefficient Ks). Plus ***Ns*** est faible et plus l'effet (le pique) spéculaire est fort. A l'inverse, plus ***Ns*** augmente et plus l'effet spéculaire s'étend et s'estompe. ***Ns*** peut prendre les valeurs de 0.0 à 1000.0.

b. Les tests de performances du Raytracing

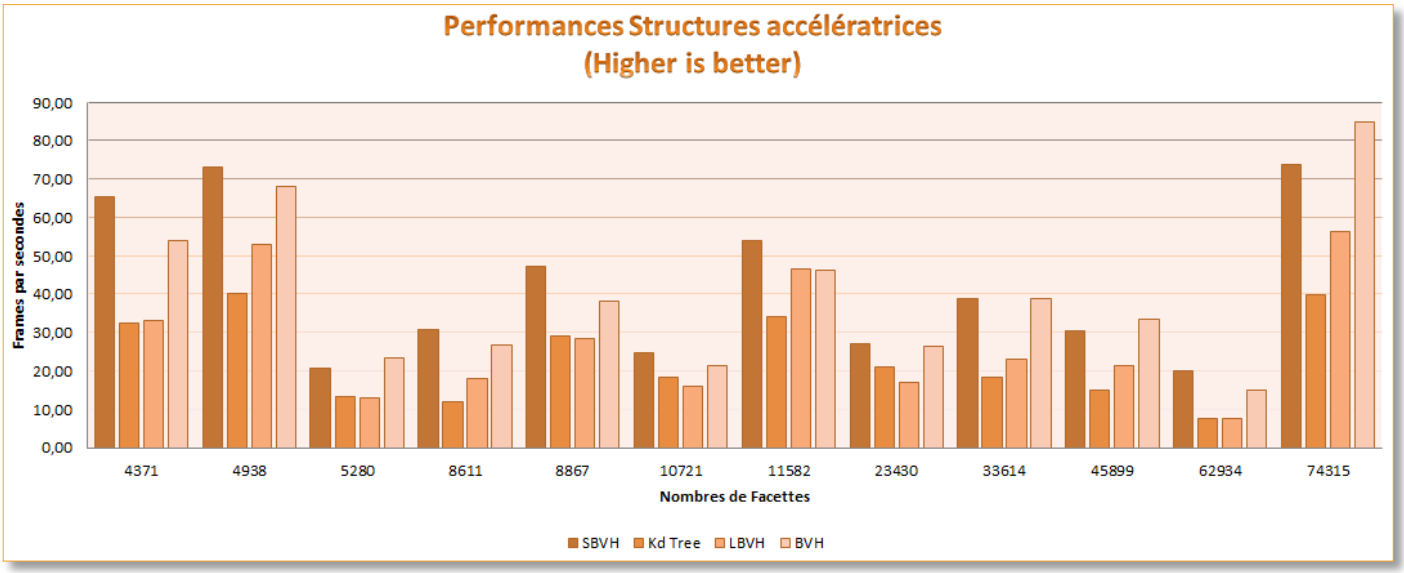
A ce stade, nous pouvons nous permettre de faire des tests de performances de notre raytracer selon la complexité de la géométrie, selon le type de rendu que l'on demande et selon la structure accélératrice utilisée. Il faut bien se rappeler que nous sommes en temps réel, la performance se mesure donc en FPS (frame par seconde) et en seconde pour la construction de l'arbre. Pour plus d'homogénéité dans le test et pour vraiment comparer les performances de chaque structure, chaque scène est rendue avec les paramètres suivants :

- ⇨ méthode de rendu : ***ambient occlusion and phong***
- ⇨ options: ***antialiasing*** et ***4 multiply ao*** (on passe 4 fois pour ambient occlusion)

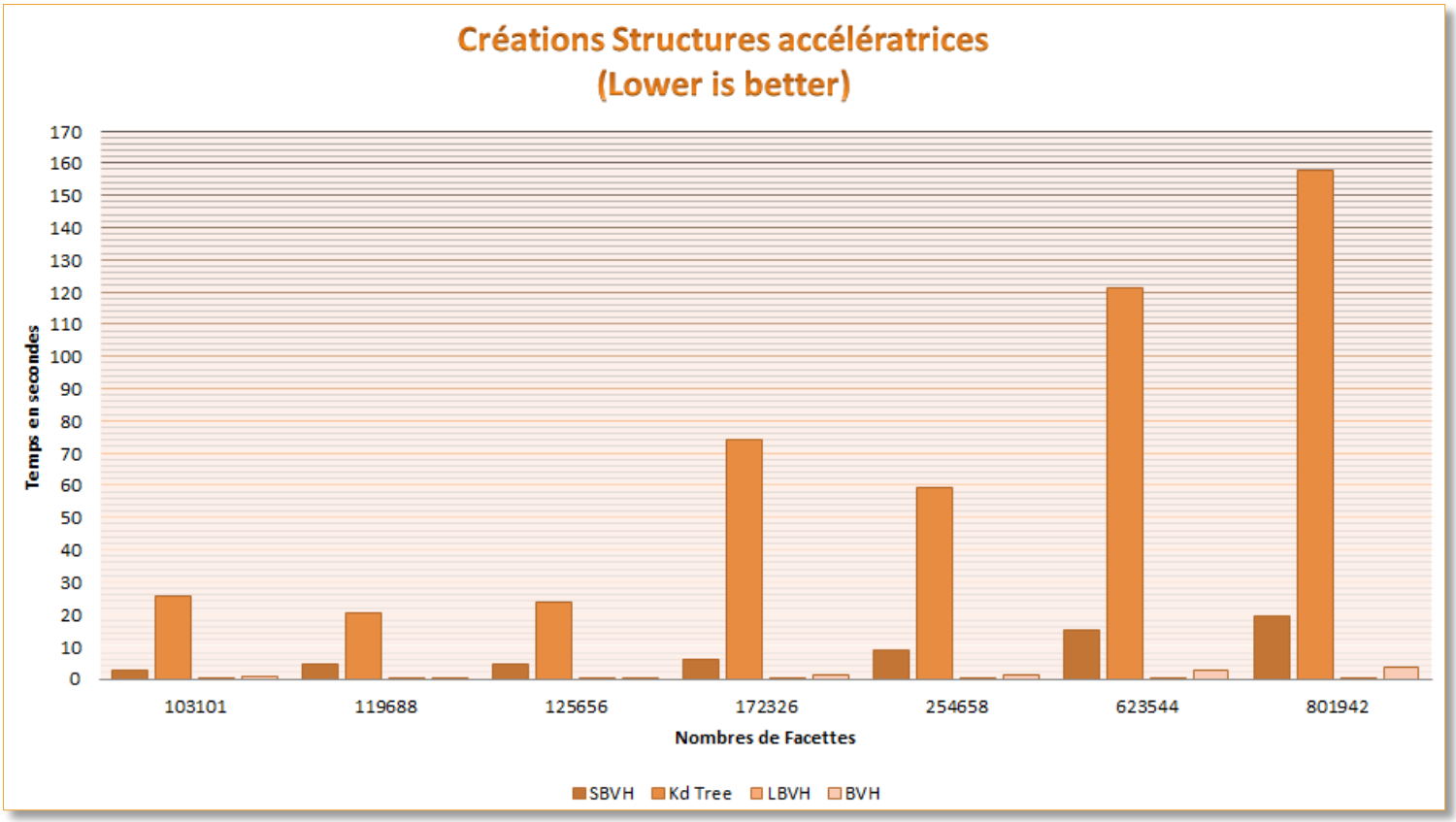
Les objets dans les tests suivants sont tous des véhicules ou avions, certes très différents les uns des autres pour avoir un test intéressant, mais on garde quand même à l'esprit que les fonctions programmées sont destinées à Crira (qui effectue dans la plus part du temps des rendus d'aéronefs).



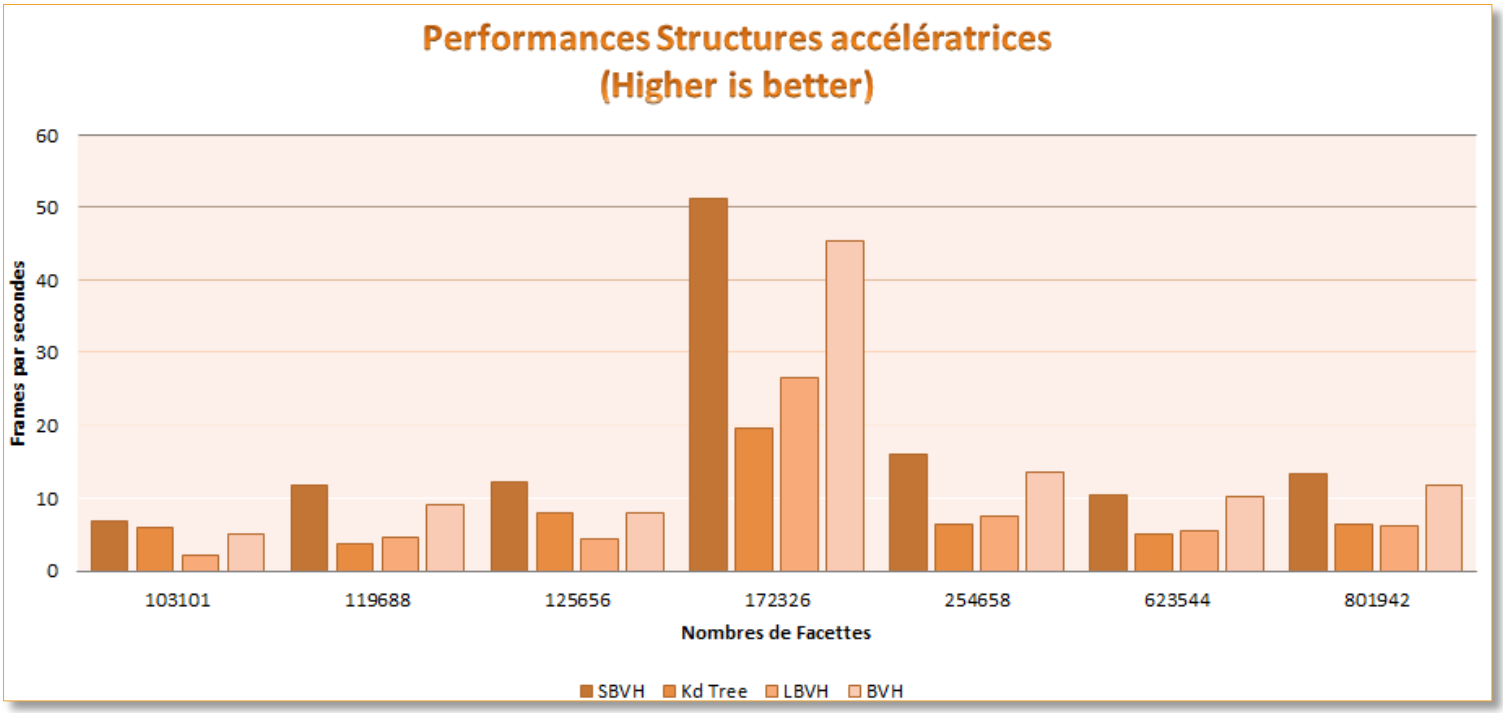
Temps de créations d'une structure accélératrice en fonction du nombre de facettes de l'objet chargé (pour de petites géométries).



Performances de rendu pour chaque structure accélératrice en fonction du nombre de facettes de l'objet chargé (pour de petites géométries).



Temps de créations d'une structure accélératrice en fonction du nombre de facettes de l'objet chargé (pour de grandes géométries).

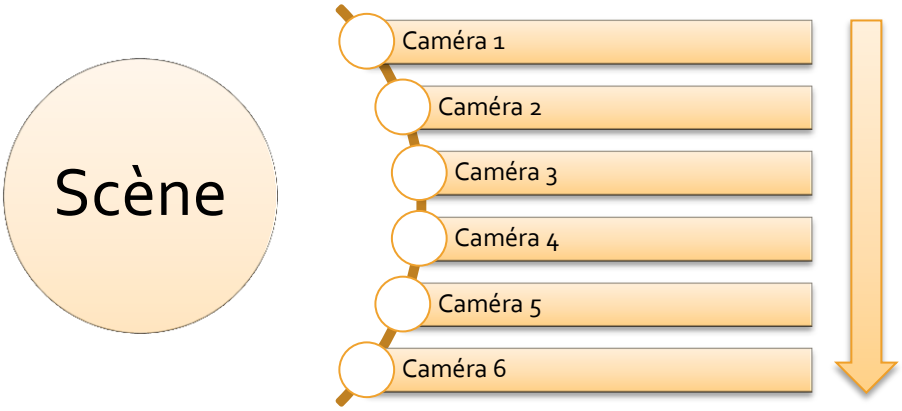


Performances de rendu pour chaque structure accélératrice en fonction du nombre de facettes de l'objet chargé (pour de grandes géométries).

Petit aspect technique

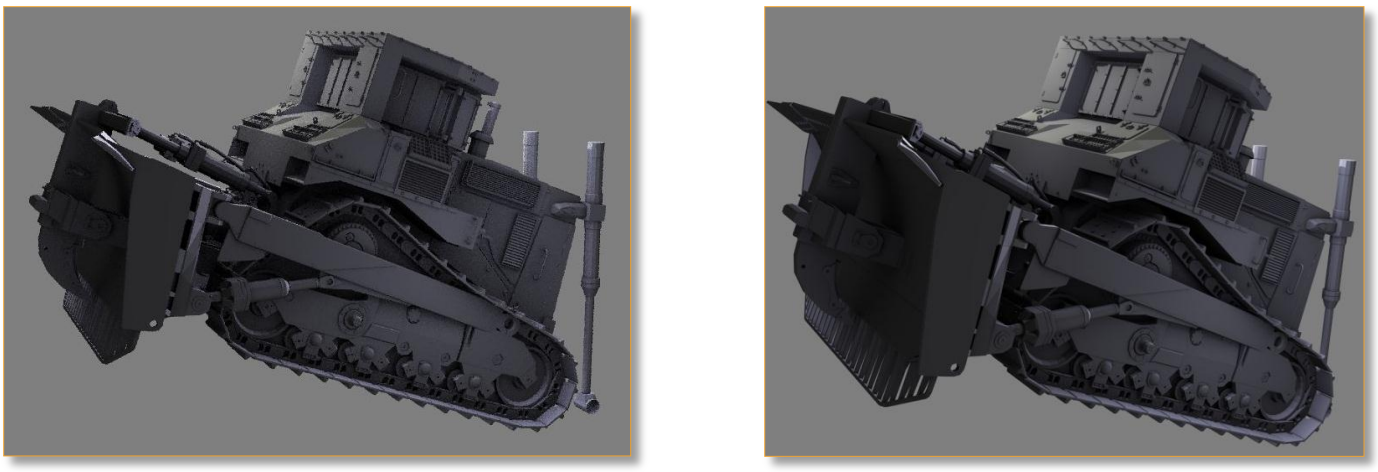
Quand on navigue dans la scène avec des options de type anti aliasing et rendu des matériaux lues dans les fichiers .mtl, on remarque bien que les performances diminuent grandement. Cela est dû au fait que des milliers de rayons sont envoyés par seconde pour calculer la scène. Pour chaque frame, on a un rayon qui récupère la couleur de l'objet, un rayon avec plusieurs rebonds qui va calculer les ombres, un rayon avec plusieurs rebonds qui va calculer les réflexions. On a donc 3 rayons multipliés par le nombre de pixels dans la fenêtre, multipliés par le nombre de frames par secondes. Donc pour une scène de 1024*768 qui tourne à 40 FPS on a donc 3*1024*768*40 = 94371840 rayons par secondes. Cette performances est obtenue grâce à la puissance de la carte graphique et du langage OPTIX / CUDA.

Autre point qui est intéressant de comprendre: *comment peut-on naviguer dans la scène sans avoir une grosse chute des performances ?*
Tout simplement grâce à ce que l'on appelle dans Optix : « ***l'accumulation de caméra*** ». Cette technique va, quand on navigue dans la scène, créer des caméras à chaque point que l'on visite et va effectuer la transition de où l'on était à où l'on va, tout en supprimant les caméras obsolètes. Pour optimiser les performances, on utilise un rendu ***progressif*** : l'image se construit au fur et à mesure du temps. On utilise ce mode de rendu pour permettre à l'utilisateur de naviguer avec fluidité dans la scène sans que le logiciel et la carte graphique aient à calculer l'image complète à chaque fois. Ces deux méthodes réunies ensemble vont donc fonctionner à la manière d'un streaming vidéo : on charge ce que l'on a besoin au fur et à mesure du temps et on supprime ce que l'on a déjà lu.



c. Les améliorations

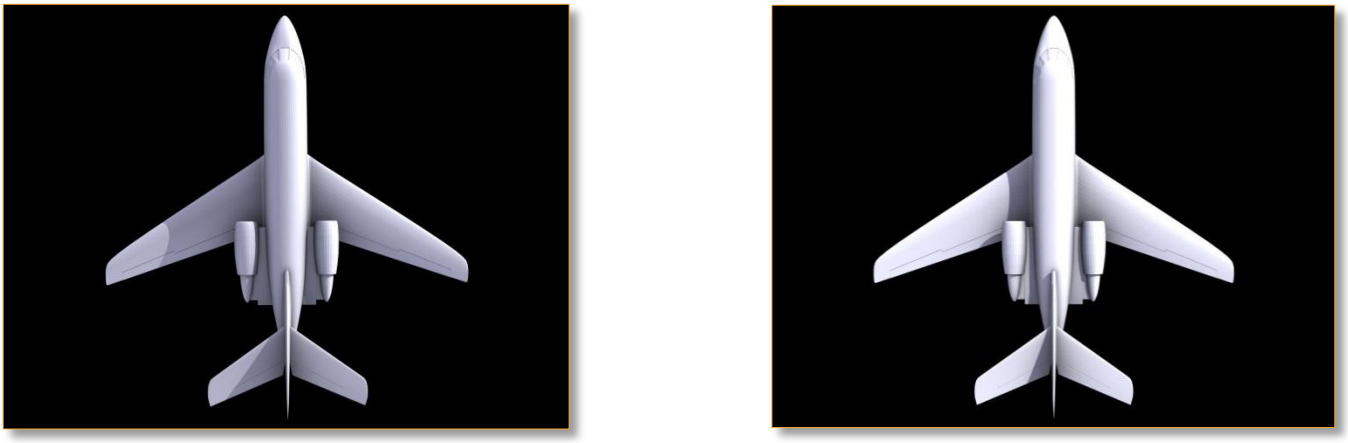
Maintenant que l'on a une base solide et un logiciel qui fonctionne correctement, j'ai ajouté des options supplémentaires dans le raytracer. Tout d'abord la possibilité de modifier le temps de rendu du mode ***progressif*** (l'image se construit au fur et à mesure du temps). Par défaut, ce temps de rendu est de 10 secondes mais grâce à cette option vous pouvez augmenter ou diminuer ce temps.



Temps de rendu de 1 seconde à gauche et de 10 secondes à droite.

Il est vrai qu'il est difficile de remarquer les différences entre ces deux images mais à taille réelle on remarque du grain et des imperfections sur l'image de gauche.

L'autre option que j'ai implémenté c'est la possibilité de modifier en temps réel la position du soleil. On modifie ces coordonnées x, y et z grâce au touche du clavier et on voit directement le changement dans la fenêtre OpenGL. Cette option est pratique car elle permet de changer l'éclairage sans toutefois recharger ou relancer le programme.



Modification de la position du soleil en temps réel sans recommencer le calcul.

La conversion automatique des images

Lorsque l'on effectue un calcul avec ***ARTGpu***, on a la possibilité de sauvegarder une ou plusieurs images du rendu effectué. Ces images sont enregistrées dans le dossier contenant les exécutables et sont au format .ppm. C'est un format de fichier sans perte mais qui n'est pas interprété nativement par windows. Pour remédier à ce désagrément, le script ***Interface.pl*** fait appelle à un script ***convert.pl*** qui va explorer le dossier contenant les exécutables, vérifier la présence de fichiers images .ppm et les convertir dans un format plus traditionnel. J'ai choisi que ce format sera le png. Après avoir converti les images, le script va couper/coller les images .ppm et .png dans un dossier de résultats qui sera créé à l'endroit où la géométrie 3d se trouve afin d'avoir toutes les données centralisées au même endroit.

La gestion des jets

Lors du choix des options, on a la possibilité de prendre en compte les **jets** des appareils. Les paramètres et aspects 3d des jets sont contenus dans le fichier ***jet_SDM*** (calculé au préalable par Crira). Ce fichier est au format .bdd est doit être converti au format .obj. Pour cela le script ***Interface.pl*** fait appel au script ***jets.pl*** qui va convertir et sauvegarder la géométrie des jets dans le fichier ***jet_SDM.obj*** au format obj.
Une fois cette opération faite et le fichier jet créé, l'utilisateur a la possibilité de choisir dans les paramètres d'utiliser un ou deux jets selon la géométrie de l'avion sélectionné. ARTGpu va ensuite charger la géométrie et l'intégrer dans la scène en respectant les paramètres.
Lors du chargement des jets, j'ai intégré dans le code source une matrice de transformation qui nous permettra de positionner correctement les jets derrières les tuyères. Pour l'instant, les jets ne se positionnement pas correctement. C'est toujours une option en développement.

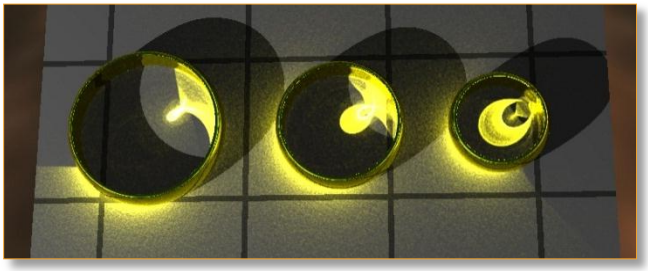
Cette option est disponible pour le raytracer et compatible avec le path tracing. Le chargement des obj se faisant de manière différente dans le photon mapping, je n'ai pas réussi à intégrer les jets à ce module.

d. Mise en place du Photon Mapping

Afin de fournir une application de démonstration complète et capable de faire des rendus de types différents, la décision a été prise d'ajouter un module de rendu en photon mapping. Pour garder une application optimisée et rapide, j'ai décidé de séparer les deux modules de rendu en deux exécutables (et donc deux projets Visual Studio) distincts. On a donc ***ARTRaytracing.exe*** pour le raytracing et ***ARTPhotonMapping.exe*** pour le photon mapping. Construire l'application de cette façon permettra dans le futur d'ajouter facilement d'autre module.

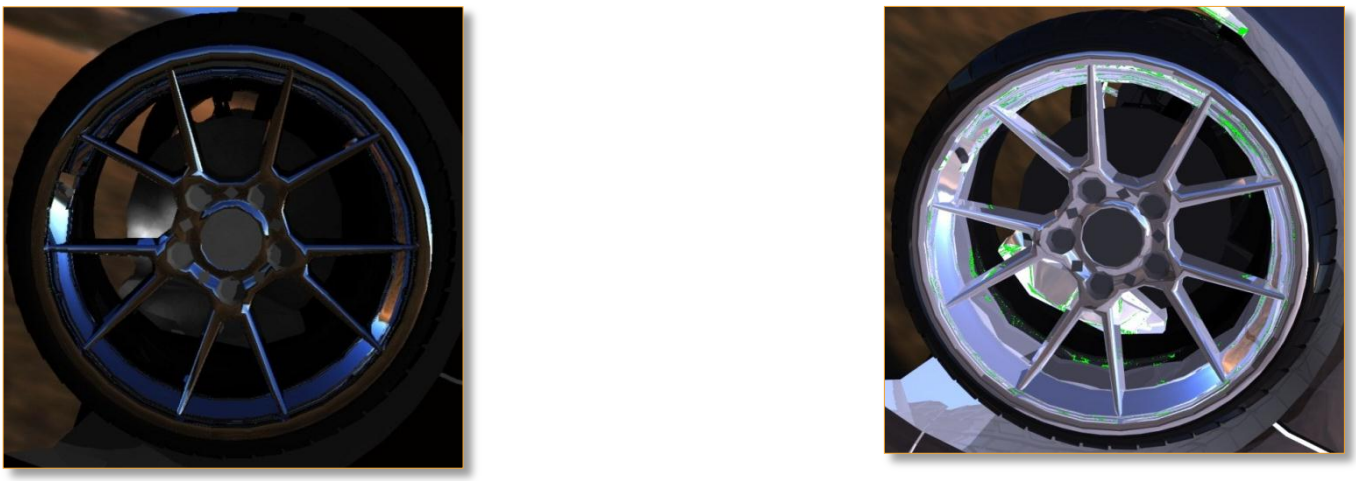
J'ai modifié le script ***Interface.pl*** afin qu'il propose à l'utilisateur quel module choisir pour le rendu. Une fois le choix entré par l'utilisateur et les options choisies, le script va exécuter le module correspondant en lui passant en paramètres (***argc*** et ***argv***) les options.
Le script ***materiaux.pl*** va, si on sélectionne le module de photon mapping, modifier automatiquement le fichier.mtl associé à la géométrie afin de le rendre compatible avec ce type de rendu (modification du paramètre illum 2 en illum 3, *explication en annexe dans la partie fichier .obj*).

Je me suis donc inspiré d'un des samples du SDK d'Optix pour développer ce module de photon mapping. Ce module récupère dans le tableau de paramètres le chemin du fichier .obj à charger, les options d'environnement (couleur unie ou environnement map), et le temps de rendu souhaité. Le module gère donc la plupart des options intégrées au raytracing (comme la modification du soleil en temps réel, le choix du temps de rendu,... je ne reviendrai donc pas sur ces aspects déjà traités). Cependant, le photon mapping fonctionne différemment car il utilise un raytracer dans son calcul. En effet, pour afficher l'objet, le module de photon mapping effectue un raytracing classique. Le photon mapping, à proprement parler, ne commence qu'ensuite car le but du photon mapping est le rendu réaliste des lumières et de l'éclairage.
Avant son démarrage, le module vous demandera plusieurs paramètres comme la puissance des lumières, la taille de la lumière et la taille des photons et ensuite il affichera l'image. On remarque bien avec ce rendu que l'un des principaux avantages du photon mapping est la gestion très réaliste des reflets grâce au support des caustiques (*voir image ci-dessous*).



Rendu d'une scène en photon mapping avec apparition de caustiques.

Cependant, on perd l'aspect temps réel car il faut laisser le temps au programme de générer assez de photons pour rendre l'image exploitable. Ci-dessous un autre exemple de rendu :

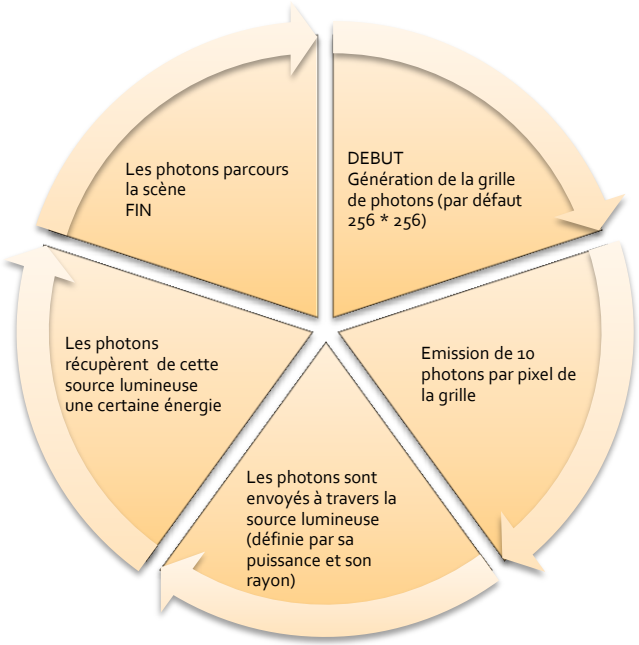


Rendu de 30 secondes en photon mapping à gauche et en raytracing classique à droite (gente spéculaire et disque de frein diffus).

On remarque assez facilement la qualité de la réflexion pour le module photon mapping avec même l'apparition d'une caustique sur le disque de frein. On voit bien que les photons rebondissent et éclairent un autre objet. Alors que pour le raytracing classique, on a une réflexion basique (uniquement spéculaire) et les objets proches (qui ne sont pas situés dans un angle de spécularité) n'apportent aucune contribution. On aperçoit même des aberrations (en vert) dues aux manques d'informations de couleurs à cet endroit de l'objet.

Petit aspect technique

Les photons sont issus d'une source lumineuse mais avant d'être issus de cette source, ils sont générés par une grille de photons. Cette grille est définie au début du programme et ne peut être modifiée par une option. Si l'on veut modifier sa taille, on doit ensuite recompiler le projet. Il en est de même pour le nombre de photons.
Donc pour une grille par défaut de 256 par 256 et pour un nombre de photons de 10, il va y avoir 10 * 256 * 256 photons envoyés par image soit pour une scène tournant à 30 FPS : 19 660 800 photons par secondes.
En effet, chaque « pixel » de la grille génère le nombre de photons définis par défaut (10 dans notre cas). Après avoir été générés, les photons passent à travers la source lumineuse (on a défini sa taille grâce à l'option Light Radius) pour enfin atteindre la scène. Chaque photon a une énergie de départ qui décroît au fur et à mesure de son parcours et de ces rebonds pour participer à la fin à l'éclairement de la scène et de l'objet.

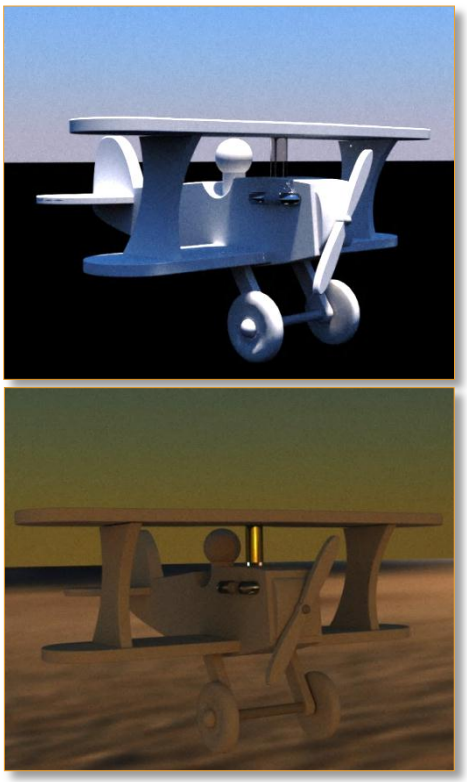

DEBUT
Génération de la grille de photons (par défaut 256 * 256)
Emission de 10 photons par pixel de la grille
Les photons sont envoyés à travers la source lumineuse (définie par sa puissance et son rayon)
Les photons récupèrent de cette source lumineuse une certaine énergie
Les photons parcours la scène
FIN

Cycle de vie des photons dans la scène.

Ce cycle est valable pour une image, nous sommes en rendu progressif, ce cycle est exécuté plusieurs fois par secondes car notre rendu est effectué à plusieurs images par secondes.

e. Mise en place du Path Tracing Markov Chain Monte Carlo

Toujours dans l'esprit d'avoir un logiciel complet qui permet de faire des comparaisons entre plusieurs techniques de rendu, il a été aussi décidé d'intégrer un module de path tracing dans ***ARTGpu***. Cependant, toujours à cause de la jeunesse du SDK Optix, il a été beaucoup plus difficile de trouver des exemples, documents et informations sur ce type de rendu. C'est un peu par hasard que je suis tombé sur un path tracing bidirectionnel avec méthode Markov Chain Monte Carlo (*cette méthode sera expliquée en détails dans le V.*). J'ai donc inséré cette technique dans un nouveau module de la même manière que pour le photon mapping. Ce module est intégré sous la forme d'un nouveau projet Visual Studio qui génère un exécutable ***ARTPathTracing.exe.***
Le script ***Interface.pl*** passe en paramètres, à ce nouveau module, les options de rendu, chemins de fichiers et choix d'environnements. L'éclairage en temps réel et le choix de l'environnement map ainsi que la gestion des jets sont similaires aux chapitres précédents et ne seront donc pas retraitées.

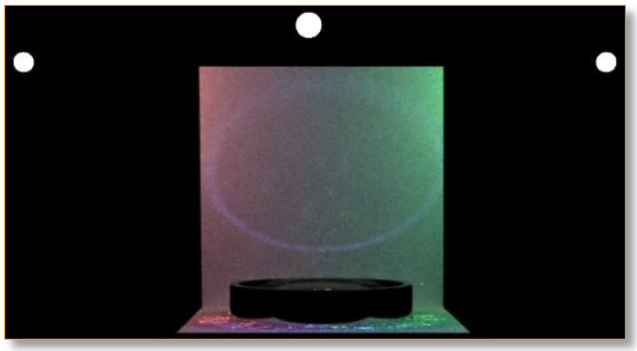


Rendu d'un objet 3d avec sunsky plus couleur unie en haut à gauche et sunsky plus environnement map ci-contre.

Cette méthode de rendu permet donc de faire l'affichage d'objet 3d .obj comme nos modules précédents mais celui-ci contient des spécificités qui lui sont propres. En effet, il génère par défaut un ciel qui est une fonction ***miss*** (*voir définition dans la partie raytracing*). Ce ciel, ou sunsky, est en fait « plaqué » sur une demi-sphère au-dessus de la bounding box (donc de l'objet qui est au centre de celle-ci). Le fonctionnement est le même que pour l'environnement map (*image de gauche*).
Cependant, selon notre point de vue, le programme va calculer la partie du ciel à afficher. Mais on peut aussi changer la position du soleil, le programme va alors calculer directement les nouvelles couleurs et ainsi simuler un coucher de soleil (dans le cas où l'on place le soleil à l'horizon) comme nous pouvons le voir sur les images ci-contre. On peut aussi associer à ce ciel soit une couleur unie, soit une environnement map qui seront dans la demi sphère inférieure de l'espace (*voir schéma explicatif ci-contre*).

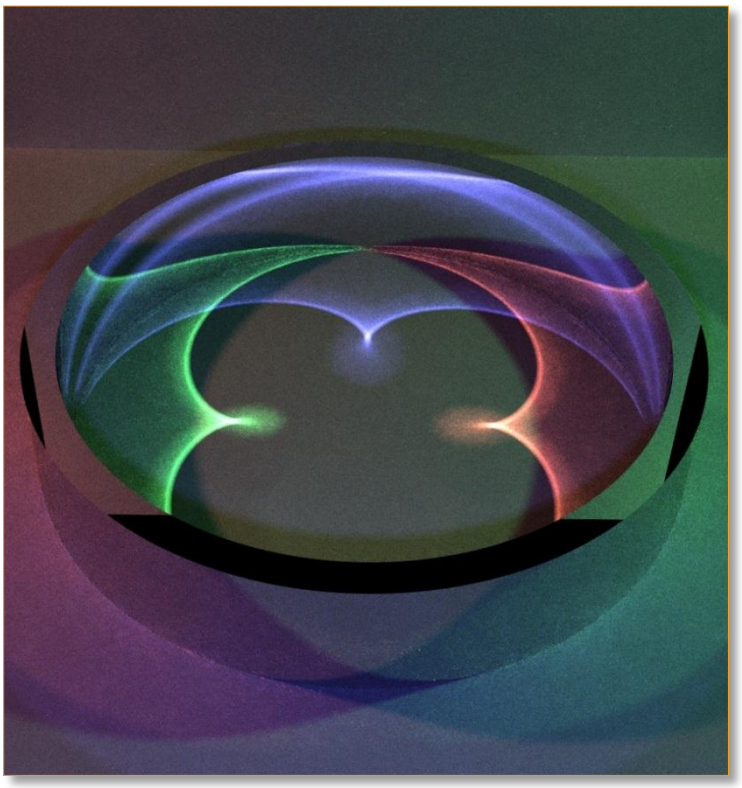
Construction de la scène.

Une autre option est la possibilité de définir des objets qui vont émettre de la lumière. Dans le photon mapping et le raytracing des lumières sont définis par défauts lors de la construction de la scène. Pour notre module de path tracing, si on ajoute le mot clef ***emissive*** dans le fichier matériaux .mtl, l'objet associé à ce matériau devient source de lumière (et le ciel n'est pas généré). Dans l'image ci-dessous, on a défini trois sphères comme source de lumière (une rouge, une verte, une bleue). Elles apparaissent blanches car c'est la couleur par défaut d'une source. Il faut bien faire la différence entre la couleur avec laquelle elle nous apparaît (coefficient ***Ka*** et ***Kd***) et la lumière qu'elle émet (coefficient ***emissive***).



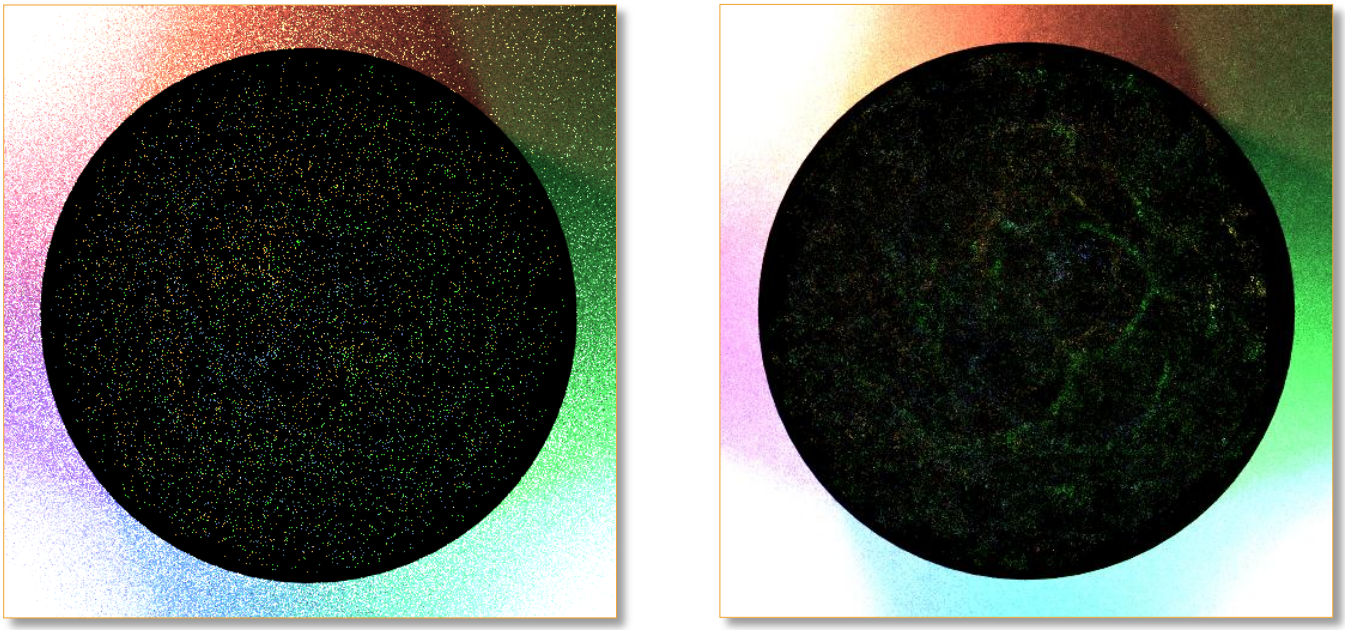
Trois sphères qui émettent de la lumière bleu, rouge et verte (définit par le coefficient emissive) et qui apparaissent blanche (définit par les coefficients Ka et Kd).

Sur cette nouvelle image, on remarque bien que notre module de path tracing gère aussi les effets caustiques, cela vient du fait que l'on utilise un path tracing bidirectionnel, mais on y reviendra plus tard.



Gestion des caustiques par le path tracing.

Enfin la dernière option intéressante est la possibilité de choisir les chemins des rayons de manières aléatoires (méthode classique du path tracing) ou d'utiliser la méthode MCMC (Markov Chain Monte Carlo) afin d'optimiser ce tirage. Cette option (*expliqué en détails dans le V.*) permet d'accélérer énormément le temps de rendu de notre path tracing.



Ci-dessus à gauche, une image générée en 10 secondes avec la méthode classique, à droite avec la MCMC.

On remarque bien que pour un même temps et une même scène, la MCMC est plus rapide et plus précise.

f. Diagrammes du projet ARTGpu

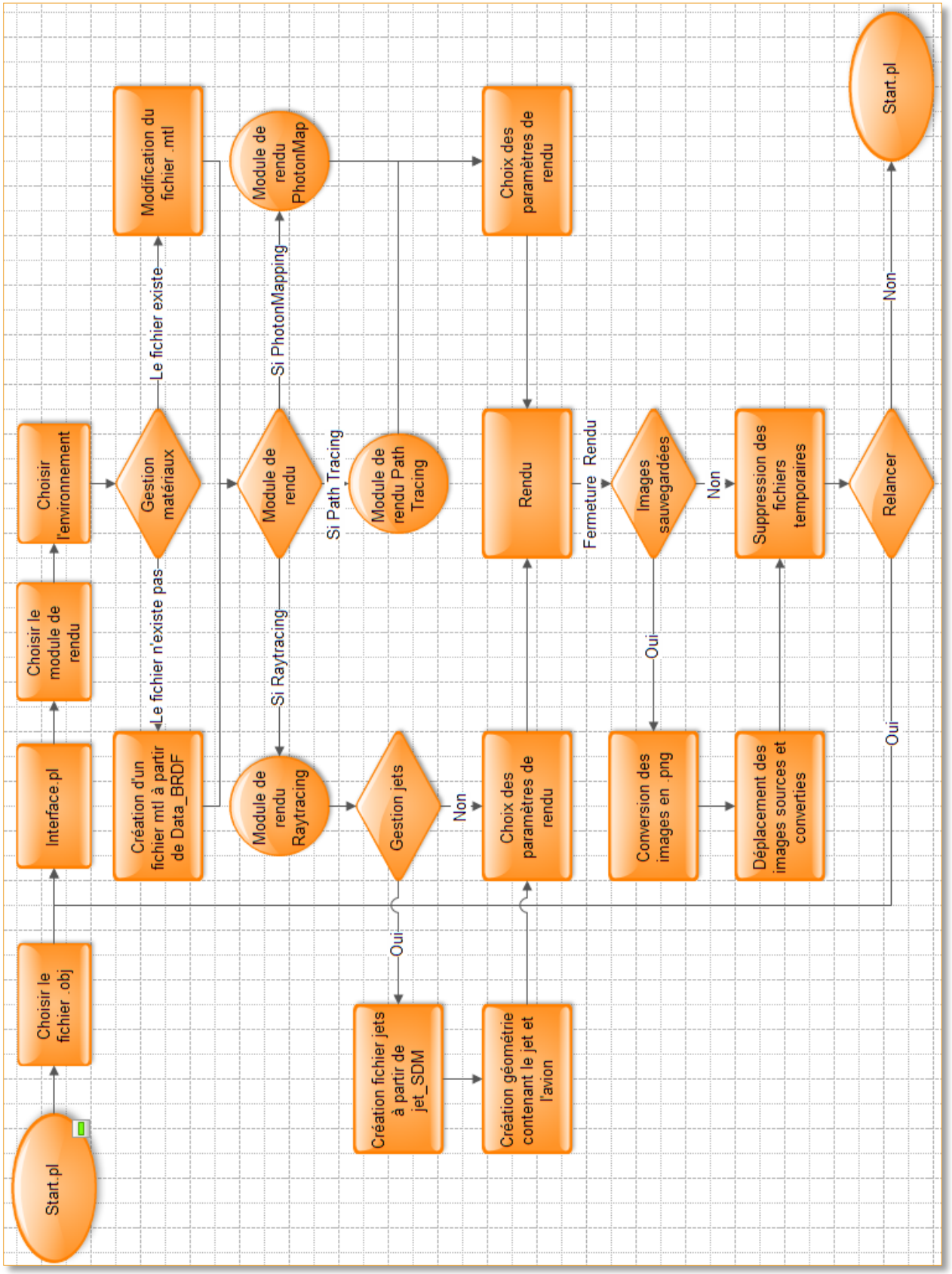


Diagramme de scénario pour ARTGpu.

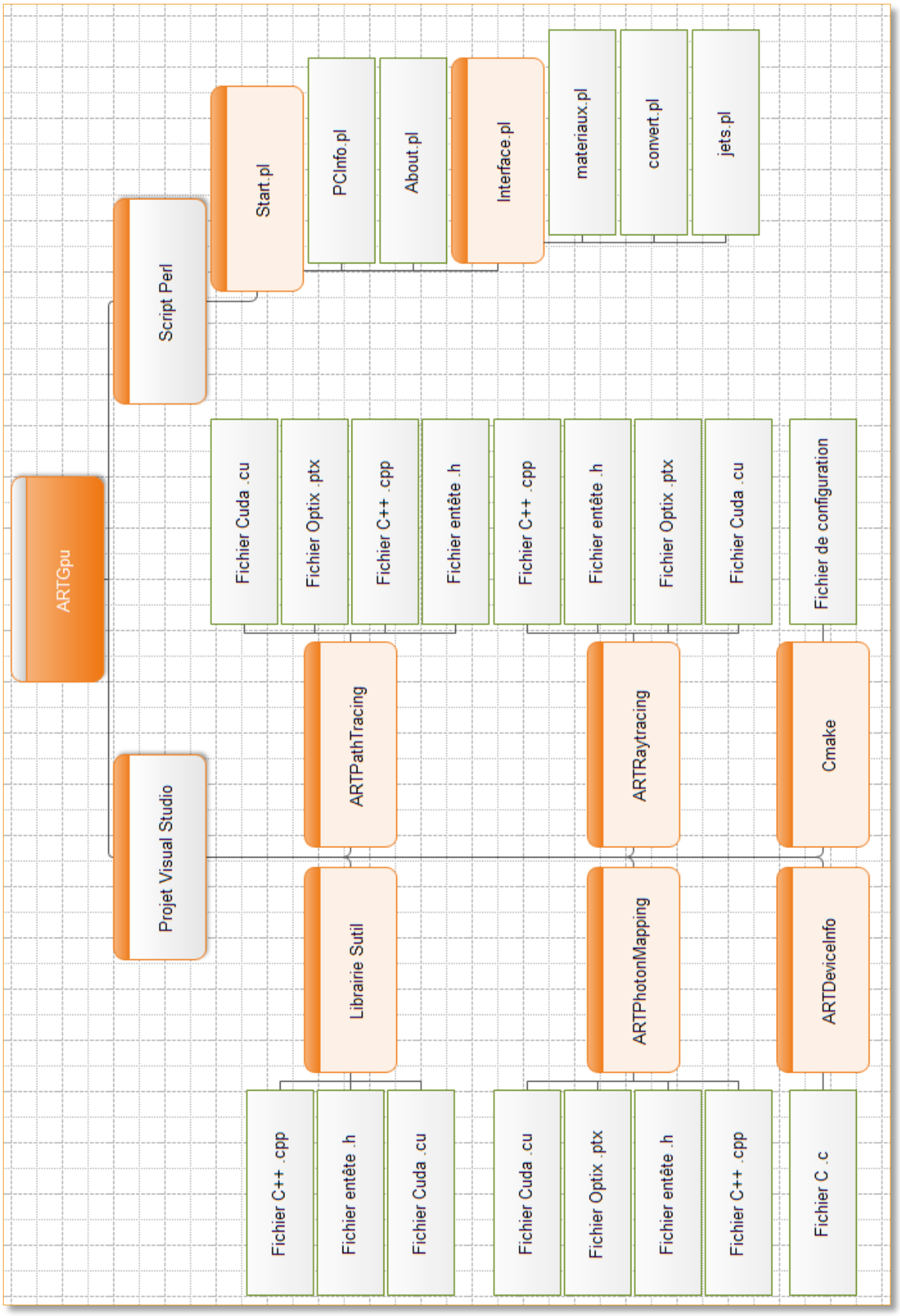


Diagramme de fichiers pour ARTGpu.

g. Fonctions communes, problèmes et solutions

J'ai ajouté deux options importantes dans notre logiciel ***ARTGpu***, cependant, ces fonctions ne fonctionnent pas comme je le voulais.

La première option est le chargement de plusieurs fichiers .obj contenant une géométrie 3D. J'ai donc modifié les codes sources afin de prendre en compte cette nouvelle possibilité. Après de nombreux jours de tests et de programmation, j'ai réussi à mettre en place ce chargement multiple pour le raytracing et pour le path tracing. Cependant pour le photon mapping, cette option ne fonctionne pas car on n'utilise pas des géométries group comme dans les autres modules. La fonction ainsi codée ne peut donc pas s'intégrer et je n'ai pas à ce jour trouvé d'astuce pour contourner ce problème. Pour charger plusieurs obj, il faut attribuer à chaque fichier obj une géométrie groupe. Lors du chargement, Optix découpe l'objet en plusieurs géométries instances comprises dans la géométrie groupe. Nous devons donc créer une géométrie groupe principale dans laquelle on intègre la géométrie groupe, et donc toutes les géométries instances, de chaque fichier obj (*Voir l'annexe sur optix pour bien comprendre la notion de géométrie groupe*).

La seconde option est l'ajout d'un repère dans la fenêtre Opengl afin de mieux se repérer dans l'espace. Pour l'intégrer, je crée des lignes Opengl avec une couleur différente chacune auxquelles je donne des positions fixes. Ensuite j'affiche ce repère dans la fenêtre Opengl (l'affichage se fait dans la fonction display de la fenêtre). Alors que cette technique est la bonne, le repère ne s'affiche pas et fausse le rendu de la géométrie (le rendu devient complètement bleu). Après des heures de recherche avec une stagiaire étudiante en Master d'Informatique spécialité image, nous remarquons que le fonctionnement d'Optix provoque ces disfonctionnement.
Voici comment fonctionne mon programme :

Chargment d'une géométrie (CPU)
Calcul du rendu (GPU)
Flux d'information sur le rendu 3D(GPU)
Conversion des informations 3D en texture 2D (CPU)
Affichage de la texture 2D dans la fenêtre Opengl (CPU)

Pour garder une puissance de calcul et éviter d'être bridé par le CPU, le GPU effectue tous les calculs et stocke ensuite les informations de couleurs pour chaque pixel de l'image. Le CPU n'a plus qu'à convertir ces tableaux de pixels en images (texture 2d) et les afficher. Cette méthode permet donc de garder le coté extrêmement parallèle du programme et d'utiliser au mieux la puissance du GPU. Cependant l'image affichée dans la fenêtre n'est pas en 3D, lorsque nous avons l'impression de naviguer dans une scène 3D, nous ne faisons en réalité qu'un déplacement en ***x*** et en ***y*** qui est ensuite interprété par Optix comme un déplacement dans la scène. Pour remédier à cela, il faut convertir l'information déplacement en ***x*** et le déplacement en ***y*** en une information d'angle et ensuite synchroniser le repère avec ce déplacement. Vu que je n'ai pas réussi à mettre en place cette synchronisation, j'ai créé un repère dans un fichier .obj et si l'on choisit l'option axe, les modules raytracing et path tracing vont charger ce repère et l'afficher. Cela reste du bricolage à améliorer.

4. Compatibilité

Notre logiciel ARTGpu fonctionne avec les versions de CUDA supérieur à la 2.3. Les versions précédentes ne prennent pas en compte Optix. De plus, pour exécuter correctement tous les modules, il faut avoir une carte graphique de type Nvidia Fermi avec une puissance de calcul (*compute support*) supérieure ou égale à 2.0.
Pour savoir si l'ordinateur peut faire tourner correctement tous les modules, un petit programme de compatibilité a été introduit dans ARTGpu. Dans le menu principale du script Start.pl, choisissez l'option 2, vous aurez alors les informations de votre système affichées et si votre machine est compatible ou non.

5. Test de performance ARTGpu

Pour voir comment ce comporte le programme, nous allons faire des tests de performances en raytracing avec les mêmes paramètres que les tests précédents (***aop aa m 4)*** et en photon mapping avec un objet de 9302 facettes.

Configuration	Raytracing (FPS)	Photon mapping (FPS)
Intel Core i7 – Nvidia Tesla C2070	95.50	10.78
Intel Xeon – Nvidia Quadro 1800	3.65	7.5

6. Intégration avec Crira

La plupart des fonctions nécessaires à Crira ont été programmées. Cependant, pour avoir une bonne intégration il fallait que ARTGpu fournisse (écrive dans un fichier) les parcours des rayons (point de départ, direction, point d'intersection,...). Pour ce faire, je devais donc modifier les fichiers .cu Cuda et ajouter une fonction d'écriture. Mais c'est là que les problèmes arrivent. Il n'est pas possible de faire de l'écriture fichier en Cuda. C'est après plusieurs jours de recherche que des développeurs de Nvidia m'ont communiqué cette information (via leur forum). J'ai alors essayé d'utiliser la fonction *cudaMemcpy(),* qui permet de rapatrier les données sur la mémoire RAM de la machine et d'utiliser une écriture classique en C++, cependant les fichiers .cu qui sont des fichiers Cuda ne sont que des intermédiaires avec l'utilisateur car le compilateur nvcc compile les .cu en .ptx (langage assembleur Optix) et Optix ne connaît pas cette fonction *cudaMemcpy().* A l'heure actuelle, il n'est donc pas possible de faire de l'écriture fichier dans Optix / Cuda et cette fonction si importante va certainement empêcher l'intégration de ARTGpu dans Crira.

7. Ouverture et évolutions futures

Comme énoncé précédemment, l'intégration est compromise. Cela est d'autant plus embêtant que je ne pouvais prévoir que cette fonction ne pourrait se programmer correctement.
Cependant, il est tout à fait possible de faire évoluer ARTGpu. Pour ma part, je pense que le plus intéressant serait d'intégrer directement dans l'outil un rendu des modèles en infrarouge et ainsi d'avoir un outil de visualisation temps réel des réflexions infrarouges. Surtout qu'ajouter cette fonction dans ARTGpu est possible et que Optix le permet très certainement.

V. Les méthodes de rendu expliquées par la physique

1. L'équation de rendu (article informatif issu d'internet)

(Source Wikipédia)

En infographie, l'équation de rendu est une équation intégrale dans laquelle le rayonnement en un point est donné par la somme des rayonnements émis et réfléchi dans un espace géométrique. Il a été introduit en par David Immel et James Kajiya en 1986. Les différentes techniques de rendu réalistes en infographie tentent de résoudre cette équation.

La base physique de l'équation de rendu est la loi de conservation de l'énergie. En supposant que ***L*** représente la radiance, on a pour chaque position de l'espace la lumière sortante ***Lo*** qui est la somme de la lumière émise ***Le*** et de la lumière réfléchie. La lumière réfléchie est la somme de la lumière entrante ***Li*** dans toutes les directions, multipliée par la réflexion de surface et le cosinus de l'angle d'incidence.

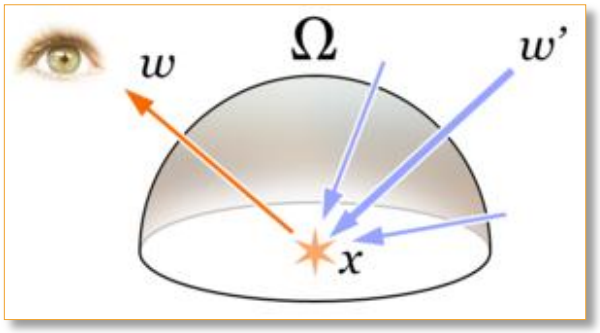


Illustration de la demi-sphère de l'espace dans laquelle l'équation s'applique (source image wikipedia).

L'équation de rendu a donc la forme suivante :

$$L_o(\mathbf{x},\omega,\lambda,t) = L_e(\mathbf{x},\omega,\lambda,t) + \int_\Omega f_r(\mathbf{x},\omega',\omega,\lambda,t) L_i(\mathbf{x},\omega',\lambda,t)(-\omega'\cdot\mathbf{n})d\omega'$$

Avec :

- ⇨ ***λ*** est une longueur d'onde particulière de la lumière.
- ⇨ ***t*** est le temps.
- ⇨ ***Lo*** est la quantité totale de lumière pour la longueur d'onde ***λ*** dans la direction ***ω*** au temps ***t*** à partir d'un point ***x.***
- ⇨ ***Le*** est la lumière émisse.
- ⇨ ***fr*** est la BRDF qui calcul la proportion de lumière réfléchie venant de ***ώ*** et se dirigeant vers ***ω*** en un point ***x***, au temps t, pour une longueur d'onde ***λ.***
- ⇨ ***Li*** est la lumière de longueur d'onde ***λ*** provenant de la direction ***ώ*** à un temps ***t.***
- ⇨ ***-ώ.n*** est l'atténuation de la lumière selon l'angle d'incidence.

On intègre sur toute la demi-sphère de l'espace.

Cependant cette équation présente plusieurs limites : elle est la même en tout point de l'espace et est donc une grosse approximation de ce qu'il se passe dans la réalité, elle ne prend pas en compte non plus la phosphorescence, la fluorescence, les interférences,.... On a donc de nombreux aspects physiques qui ne sont donc pas pris en compte.

Précision sur les coefficients Ka, Kd, Ks

En optique, Le **coefficient d'absorption** est défini par le rapport entre l'absorbance et la longueur du chemin optique parcouru par un rayonnement électromagnétique dans un milieu donné (exprimé en m^{-1} ou en cm^{-1}). Cependant en informatique et surtout en rendu d'image de synthèse le coefficient d'absorption, noté ***Ka***, est une couleur définit dans un fichier matériau .mtl et codé par trois valeurs qui correspondent au rouge, vert, bleu. Il en est de même pour le coefficient de diffusion ***Kd***.
Il faut bien avoir cette différence en tête pour comprendre les prochaines parties qui vont parler de la physique derrière les différentes techniques de rendu (raytracing, path tracing,...).

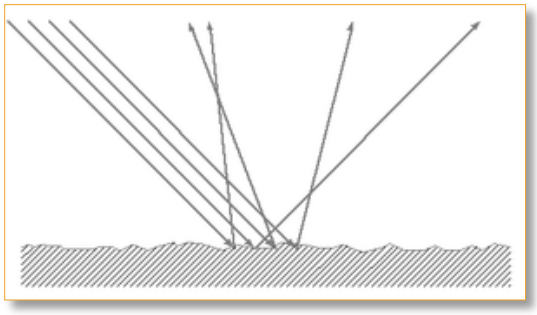
2. La physique du raytracing (article informatif issu d'internet)

(Source Wikipédia)

Un raytracer est un programme informatique mais de nombreuses notions physiques sont nécessaires pour bien comprendre son fonctionnement.

Réflexion optique

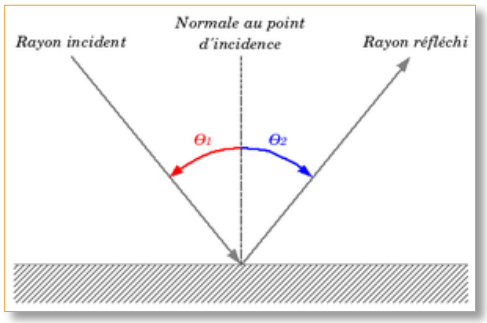
Lorsque la lumière arrive sur une interface entre deux milieux elle peut être soit réfléchie, réfractée ou absorbée. Quand on parle de réflexion, il s'agit de l'ensemble des rayons qui émergent de l'interface du côté d'origine des rayons. La réflexion de la lumière peut être **spéculaire** ou bien **diffuse** suivant la nature de l'interface. La réflexion spéculaire est la plus simple à mettre en œuvre car elle fait appel à des notions géométriques simples, contrairement à la réflexion diffuse qui nécessite des modélisations plus complexes.



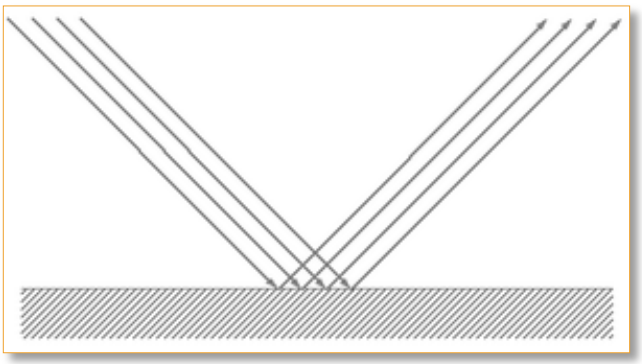
Réflexion diffuse de la lumière (source image wikipedia).

Le faisceau incident arrive sur une surface irrégulière (*image ci-contre*), comme toutes les surfaces planes qui n'ont pas été polies comme une vitre ou un miroir. La lumière est alors réfléchie dans un grand nombre de directions. C'est ce qu'on appelle une réflexion diffuse.

Le faisceau de lumière arrive sur une surface régulière (*image ci-dessous*). Les rayons sont réfléchis tous de la même façon sur la surface plane. C'est ce que l'on appelle la réflexion spéculaire. Idéalement, l'énergie du rayon incident se retrouve totalement dans le rayon réfléchi, en pratique une partie de l'énergie peut être absorbée ou diffusée au niveau de l'interface.



Loi de Snell-Descartes (source image wikipedia).



Réflexion spéculaire de la lumière (source image wikipedia).

Cette réflexion spéculaire utilise la loi géométrique mis en place par Snell-Descartes. La définition de la loi est la suivante :

- ⇨ le rayon incident et le rayon réfléchi sont dans le plan d'incidence
- ⇨ i = r

Réfraction optique

Lorsque la lumière passe d'un milieu transparent à un autre, elle est déviée. La lumière est dite "réfractée" et la propriété qui caractérise les différents milieux transparents est la "réfringence", qui se traduit par une valeur numérique : « l'indice de réfraction ».

Comme on la vue, chaque milieu transparent est caractérisé par son indice de réfraction noté n_i. On appelle dioptre la surface séparant les deux milieux. La loi de la réfraction de Snell et Descartes permet de rendre compte du phénomène. La loi est la suivante : les angles d'incidence et de réfraction (θ_1) et (θ_2), mesurés par rapport à la normale sont tels que :

$$n_1 \sin(\theta_1) = n_2 \sin(\theta_2).$$

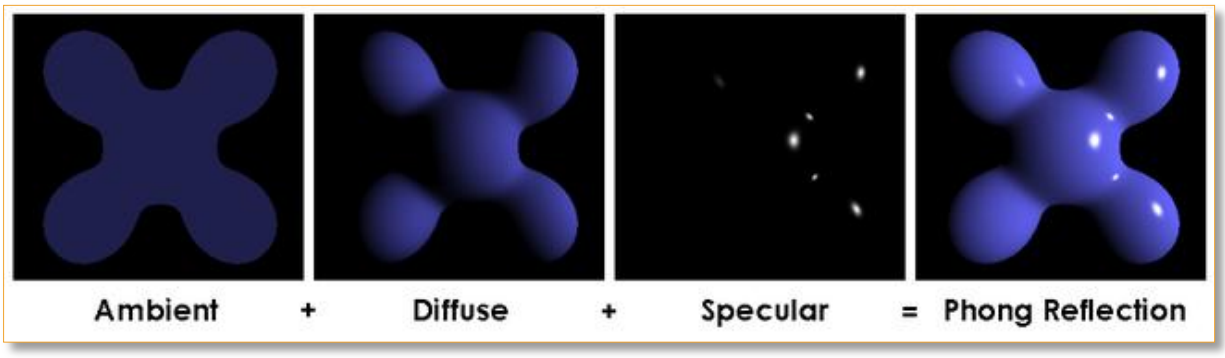
a. Le modèle de Phong

Les modèles sont là pour faire le lien entre la physique et l'informatique. Chaque modèle a des avantages et des inconvénients car ils font des approximations des règles de la physique. Nous allons nous intéresser au modèle de Phong qui est celui mis en œuvre dans notre logiciel ARTGpu.

Description du modèle

Le modèle de Phong est un modèle dans lequel le calcul se fait en chaque point. Il permet de calculer de manière précise la lumière réfléchie par le point étudié, pour cela il combine trois éléments : la lumière ambiante, la lumière diffuse et la lumière spéculaire.
L'objectif est de récupérer l'intensité lumineuse du point étudié. Pour cela, la lumière est séparée en trois composantes :

- ***La composante ambiante*** représente les parasites provenant d'autre chose que la source considérée, la lumière réfléchie par d'autres points par exemple. La lumière ambiante est supposée égale en tout point de l'espace.
- ***La composante diffuse*** indique l'intensité du rayon réfléchi qui repart en tenant compte de l'inclinaison avec laquelle la lumière incidente arrive sur la surface, mais en supposant que l'intensité est la même quelle que soit la direction que prend le rayon réfléchi.
- ***La composante spéculaire*** tient compte du fait qu'il y a plus de lumière renvoyée dans la direction de la réflexion géométrique (celle dans laquelle repartirait le rayon en arrivant sur un miroir).

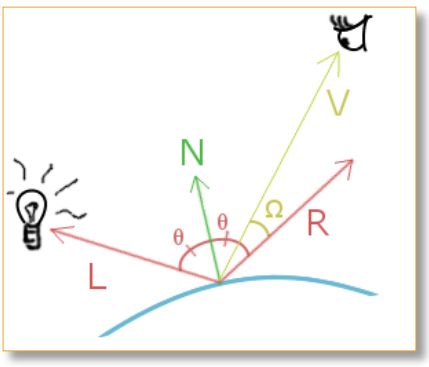


Composantes constituant un matériau dans le modèle de phong (source image wikipedia).

Définitions des paramètres et des constantes

Dans le modèle de Phong, chaque matériau a des constantes qui lui sont propres et qui correspondent au trois composantes de la lumière :

- $ka \in [0,1]$: constante liée à la composante ambiante, la proportion de lumière renvoyée.
- $kd \in [0,1]$: constante liée à la composante diffuse.
- $ks \in [0,1]$: constante liée à la composante spéculaire.
- $\alpha \gg 1$: constante liée à la *brillance* du matériau (définit la largeur du lobe autour du spéculaire).



Plus ***α*** est grand, plus la surface est brillante. Cette constante peut prendre des valeurs élevées : 10, 100 ou plus.
On appelle ***ia***, ***id*** et ***is*** l'intensité des lumières incidentes ambiante, diffuse et spéculaire. ***Ia***, ***Id***, ***Is*** et ***I*** les intensités réfléchies, ***I*** étant le total.
On définit les vecteurs suivants : $\vec{L}$ pour la lumière, $\vec{N}$ pour la normale à la surface, $\vec{V}$ pour la direction de vue de l'observateur et $\vec{R}$ pour la direction dans laquelle serait réfléchie la lumière sur un miroir.
$\vec{R}$ se déduit par la relation :

$$\vec{R} = 2.\left(\vec{N}.\vec{L}\right)\vec{N} - \vec{L} = 2\cos\theta\ \vec{N} - \vec{L}$$

Tous ces vecteurs doivent être normalisés pour que les produits scalaires donnent juste le cosinus de l'angle entre les vecteurs.

Formules

La composante ambiante, ***Ia*** est simplement donnée par : $\mathrm{Ia} = \mathrm{ia} * \mathrm{ka}$

La composante diffuse, ***Id*** est donnée par : $\mathrm{Id} = \mathrm{id} * \mathrm{kd}\ (\underset{\mathrm{L}}{\rightarrow} . \underset{\mathrm{N}}{\rightarrow}) = \mathrm{Id} * \mathrm{kd}\ \cos\theta$

Id est maximale lorsque la lumière arrive sur la surface selon la direction normale, c'est-à-dire $\underset{\mathrm{L}}{\rightarrow} = \underset{\mathrm{N}}{\rightarrow}$

La composante spéculaire, ***Is*** est donnée par : $\mathrm{Is} = \mathrm{is} * \mathrm{ks}\ (\underset{\mathrm{L}}{\rightarrow} . \underset{\mathrm{N}}{\rightarrow})^{\propto} = \mathrm{is} * \mathrm{ks}\ \cos\omega^{\propto}$

Si $\underset{\mathrm{V}}{\rightarrow} . \underset{\mathrm{R}}{\rightarrow}$, ***Is*** est maximale. Plus ***α*** est grand plus les tâches lumineuses seront petites.

L'intensité réfléchie totale est ainsi : $\mathrm{I} = \mathrm{Ia} + \mathrm{Id} + \mathrm{Is}$

Pour plusieurs sources lumineuses, on obtient comme formule complète :

$$\mathrm{I} = \mathrm{ia} * \mathrm{ka} + \sum_{\mathrm{n}\in\{\mathrm{sources}\}} (\mathrm{id,n} * \mathrm{kd}\ (\underset{\mathrm{Ln}}{\rightarrow} . \underset{\mathrm{N}}{\rightarrow}) + \mathrm{is,n} * \mathrm{ks}\ (\underset{\mathrm{Rn}}{\rightarrow} . \underset{\mathrm{V}}{\rightarrow})^{\propto})$$

Application à l'infographie

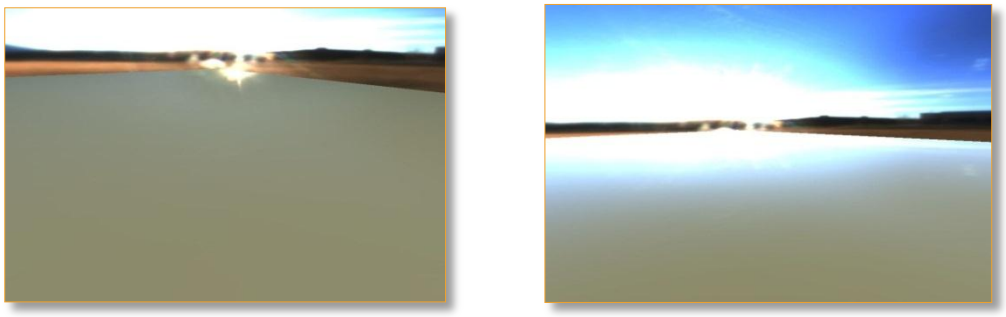
Dans le logiciel ARTGpu les coefficients ka, kd, ks vont contenir les informations de couleur en RGB (de 0 à 255 pour les trois composantes rouge, vert, bleu). C'est grâce à ces trois coefficients que l'on peut utiliser des matériaux sur les objets. Il faut aussi avoir en tête que ce modèle empirique et n'est fondé sur aucune théorie physique, mais uniquement sur les observations de Phong. Ce modèle ne prévoit pas la diffusion de la lumière avec la distance.

b. Le modèle de Fresnel

Le modèle de Fresnel est un modèle de Phong qui ne gère pas les réflexions de la même manière que le modèle de base. Alors que pour Phong, la réflexion est une réflexion spéculaire classique, pour Fresnel, la réflexion va être différente selon le point de vue de l'observateur et la longueur d'onde de la lumière. Tout comme en électromagnétisme, le coefficient de Fresnel dépend de la fréquence de l'onde (donc de sa longueur d'onde) et des angles d'incidences.

$$r = \frac{E_r}{E_i}$$

Nous n'utilisons ici que le terme de réflexion, ***Er*** champ réfléchit, ***Ei*** champ incident. La réflexion visible sera donc beaucoup plus réel car différente selon notre position par rapport à l'objet et selon la couleur de la source.



A gauche la réflexion dans le modèle de Phong, à droite la réflexion avec le modèle de Fresnel.

3. La physique du Photon Mapping

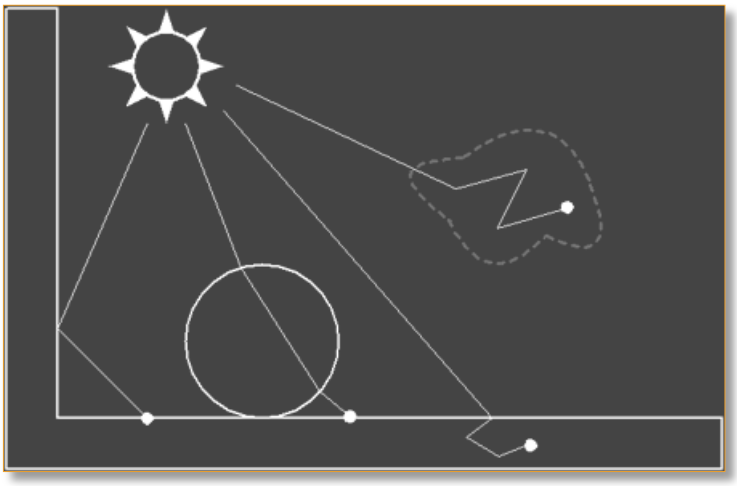
Comme définit dans la partie II, le **photon mapping** est un algorithme en deux passes. Nous allons voir dans la suite comment fonctionne ces deux passes.

First Pass : Photon Tracing

Le photon tracing est le processus d'émission de photons par la source de lumière et de tracé de ces photons dans la scène. Le but premier de cette pass est de générer la carte des photons qui sera utilisée dans la pass de rendu pour calculer les réflexions et l'illumination des surfaces.

Photon Emission : La vie d'un photon commence à la source de la lumière. Pour chaque source de lumière on définit un nombre de photon et une puissance. Plus la source est lumineuse plus elle envoie de photons.

Photon Scattering : Lorsque le photon est émis de la source, il voyage dans la scène (*voir schéma ci-contre : parcours des photons dans la scène*) et peut être soit absorbé, soit perdu. Quand un photon rencontre un objet, on peut décider combien d'énergie est absorbé, combien est reflété, combien est réfracté (grâce au paramètre de surface des matériaux de l'objet). En plus, lors de l'intersection, c'est un processus aléatoire type roulette russe qui va décider si le photon est absorbé, réfléchi, réfracté.



Photon Storing : La photon map est stockée dans un kd-tree.

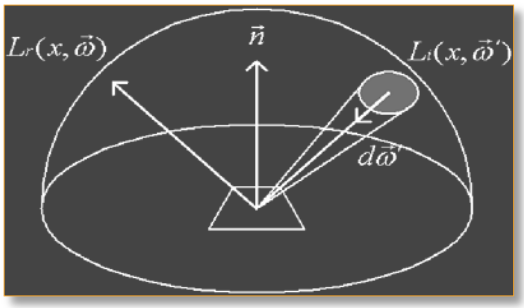
Second Pass – Rendering

Pendant la seconde pass, il va falloir résoudre l'équation de rendu approximative afin de calculer l'illumination réfléchie sur les surfaces.

$$Lr\left(x, \vec{\omega}\right) = \int fr(x, \vec{\acute{\omega}}, \vec{\omega}) Li\left(x, \vec{\acute{\omega}}\right) \cos\theta_i d\vec{\acute{\omega}}$$

Avec :

- ⇨ ***x*** un point de la surface
- ⇨ ***ώ*** la direction de la radiance entrante
- ⇨ ***ω*** la direction de sortie
- ⇨ ***fr*** la BRDF
- ⇨ ***Li*** la luminance incidente au point ***x***
- ⇨ ***cosϑi*** la projection de la surface par l'angle solide
- ⇨ ***dώ*** l'angle solide
- ⇨ ***Ω*** le demi-hémisphère au-dessus de ***x***



Dans l'équation ci-dessus, on intègre sur tous l'hémisphère supérieur la radiance entrante par l'angle solide vu par ***x***.

On peut utiliser une densité approchée pour estimer l'illumination en interrogeant les plus proches N photons de la carte de photons. Cette estimation nous donne le flux entrant.

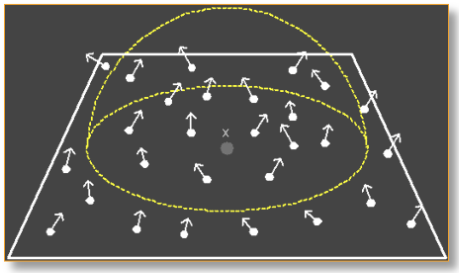


Schéma représentant la zone des N plus proches voisins.

La luminance nous donne approximativement combien de lumière est arrivée sur un surface donnée. Autrement dit, la luminance incidente est la somme de tous les photons, ***Φi***, qui est arrivée sur une unité de surface.

$$Ei = \sum_{i=0}^{n} \frac{\Phi_i}{A}$$

Avec un angle solide différent, ***dώ***, la quantité élémentaire d'éclairement, ***dEi***, peut être exprimée en fonction de la luminance.

$$dEi\left(\vec{\omega}\right) = Li\left(\vec{\omega}\right)\cos\theta_i d\vec{\omega} \text{ avec } Li\left(\vec{\omega}\right) = \frac{dEi\left(\vec{\omega}\right)}{\cos\theta_i d\vec{\omega}}$$

On peut donc substituer ***Li*** dans l'équation de rendu:

$$Lr\left(x,\vec{\omega}\right) = \int fr(x,\vec{\omega},\vec{\omega})\frac{dEi\left(\vec{\omega}\right)}{\cos\theta_i d\vec{\omega}}\cos\theta_i d\vec{\omega}$$

Et ainsi obtenir:

$$Lr\left(x,\vec{\omega}\right) = \int fr(x,\vec{\omega},\vec{\omega})dEi\left(\vec{\omega}\right)$$

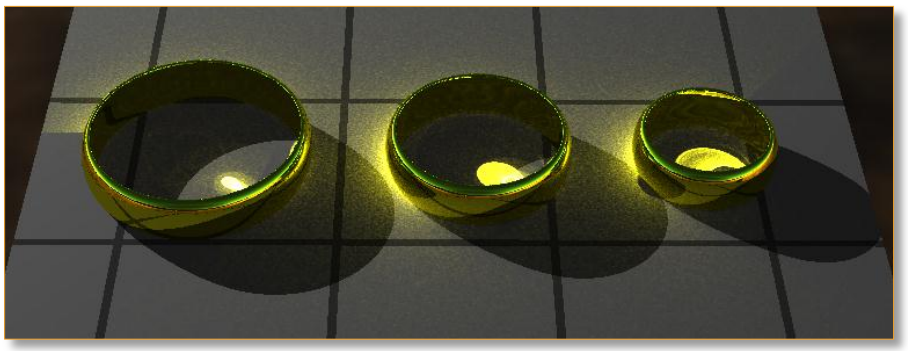
En utilisant les informations sur les ***N*** photons voisins d'une surface ***A*** on peut faire l'approximation suivante :

$$Lr\left(x,\vec{\omega}\right) = \sum_{p=1}^{n} fr(x,\vec{\omega p},\vec{\omega})\frac{\Delta\Phi_p(x,\vec{\omega p})}{\pi r^2}$$

Pour avoir une estimation correcte de la luminance, il faut avoir un nombre suffisant de photons.

Photon Mapping et Raytracing

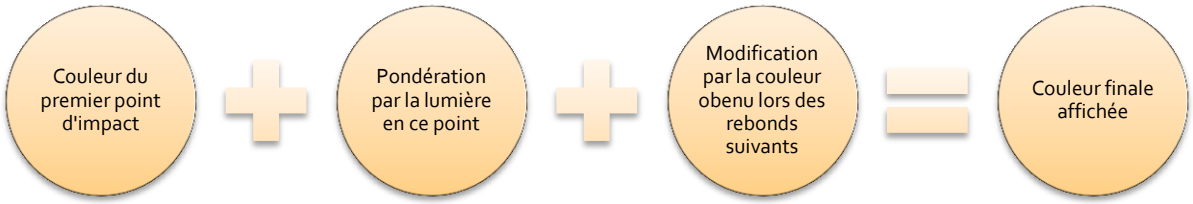
Dans le module photon mapping de notre logiciel ***ARTGpu***, nous couplons un raytracing classique avec du photon mapping. Cela permet d'avoir un rendu rapide de la géométrie dans un premier temps. Et, une fois que le rendu de la géométrie est terminé, le photon mapping rentre en action afin de simuler avec plus de réalisme le rendu de la lumière sur l'objet.



Rendu effectué avec le module photon mapping.

4. La physique du Path tracing et MCMC

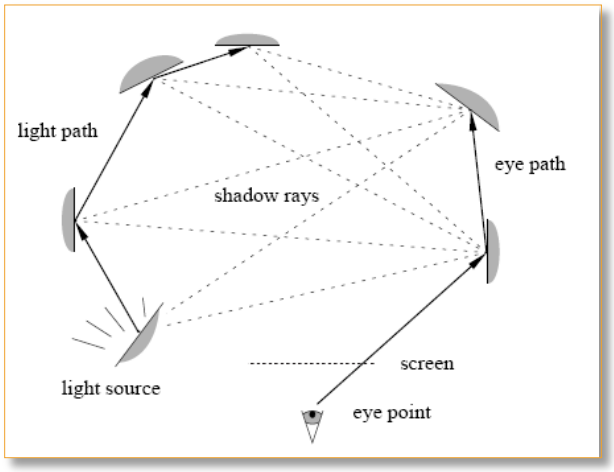
Comme on l'a vu dans le II. la technique du path tracing est une méthode de rendu d'images 3D qui consiste à lancer des rayons dans des directions aléatoires depuis l'objectif de la caméra, jusqu'à ce que ces rayons atteignent une surface géométrique. À chaque impact, le rayon va rebondir une nouvelle fois, en perdant de l'énergie à chaque rebond, jusqu'à atteindre la couleur noire ou une source lumineuse (la loi de probabilité qui définit la direction après les rebonds est définie par une pdf : probability density function). La luminosité et couleur du tout premier point d'impact sera donc une pondération de la lumière et la couleur de tous les rebonds qui ont eu lieu pour un rayon. Elle fonctionne dans les grandes lignes comme un raytracing mais la différence se fait lors du rebond et de la récupération de l'information de couleur cependant les formules sont les mêmes.


Couleur du premier point d'impact
Pondération par la lumière en ce point
Modification par la couleur obenu lors des rebonds suivants
Couleur finale affichée

Calcul de la couleur à afficher.

Le Path Tracing bidirectionnel

Le path tracing bidirectionnel est une variante, qui, en plus d'envoyer des rayons depuis la caméra, en envoie également depuis les sources lumineuses de la scène ce qui rejoint la méthode du photon mapping (je ne reviendrai pas sur l'aspect physique car il est expliqué ci-dessus).
A chaque intersection d'un rayon avec une surface, on va disperser aléatoirement les rayons vers les autres surfaces de la scène. Ensuite on va connecter tous les types de rebonds avec les rayons d'ombres afin de s'assurer qu'ils sont mutuellement visibles. Le path tracing bidirectionnel est donc une méthode qui combine les avantages du pathtracing (qui est une méthode précise) et du photon mapping (caustique et comportement des photons).



Parcours des rayons dans la scène (source http://graphics.ucsd.edu).

Monte Carlo

Le terme **méthode de Monte-Carlo** désigne toute méthode visant à calculer une valeur numérique en utilisant des procédés aléatoires. Le nom de ces méthodes, qui fait allusion aux jeux de hasard pratiqués à Monte-Carlo, a été inventé en 1947 par Nicholas Metropolis.
Les méthodes de Monte-Carlo sont particulièrement utilisées pour calculer des intégrales en dimensions plus grandes que 1 (en particulier, pour calculer des surfaces et des volumes). Elles sont également couramment utilisées en physique des particules, où des simulations probabilistes permettent d'estimer la forme d'un signal ou la sensibilité d'un détecteur. La comparaison des données mesurées à ces simulations peut permettre de mettre en évidence des caractéristiques inattendues, par exemple de nouvelles particules.

Définition : Les chaines de Markov

On appelle chaine de Markov une suite de variables aléatoires (X_n) à valeurs dans un espace probabilisé (E,B,**P**) telle que, pour chaque n, connaissant la valeur de X_n, X_{n+1} soit indépendante de X_k, pour k inférieur ou égal à n-1. Autrement dit, pour tout n et pour toutes valeurs possibles i_0, ..., i_n,i_{n+1}, la probabilité que X_{n+1} prenne la valeur i_{n+1} sachant que $X_0=i_0$,$X_1=i_1$,... ,$X_{n-1}=i_{n-1}$ et $X_n=i_n$ ne dépend que de i_{n+1} et de i_n, c'est-à-dire :

$$\mathbf{P}(X_{n+1} = i_{n+1} | X_1 = i_1, \ldots, X_n = i_n) = \mathbf{P}(X_{n+1} = i_{n+1} | X_n = i_n).$$

L'ensemble E où la suite (X_n) prend ses valeurs s'appelle l'espace des états de la suite de Markov, chaque élément de E étant un état possible de la chaine.

Markov Chain Monte Carlo

L'idée de base dans l'algorithme MCMC est de définir une chaine de Markov qui est intégrée dans une distribution stationnaire, ce qui signifie que si le système tourne pendant un temps assez long, la distribution des états visités sera proportionnelle à la distribution désirée.
Définir une chaine de Markov revient à définir une règle de transition : un processus pour sélectionner un nouvel état x+ aléatoirement qui dépend seulement de l'état x précédent. Metropolis a mis au point une technique qui prend une règle de transition qui peut ne pas produire la distribution ***π(x)*** souhaitée mais qui la transforme en une que l'on accepte.
Si on a une distribution ***T(x, x')***, les règles de transition de Metropolis opère en deux étapes :

1. Choisir x' en accord avec la distribution ***T(x, x')***
2. ***x+ = x'*** avec une probabilité ***min(1, π(x')/π(x)*** sinon ***x+ = x***

A l'étape 1, on propose x' comme état suivant. A l'étape 2, cet état est accepté avec une probabilité π(x')/π(x) (appelée probabilité d'acceptation) sinon la transition est rejetée et on recommence a première étape.

Si la chaine de Markov est capable de passer de n'importe quel états du domaine à un autre en utilisant un nombre limité de pas, on définit cette chaine comme ergodique et on est assuré de converger vers l'équilibre.
Cette algorithme de Metropolis fonctionne uniquement quand ***T(x, x') = T(x',x).***
En 1970, Hastings proposa une nouvelle probabilité d'acceptation : ***r(x, x') = min {1, π(x') T(x', x) / π(x) T(x, x') }.***
C'est cet algorithme qui est la base de la méthode de rendu MCMC.

Dans le contexte d'un rendu graphique, l'espace des états est l'espace de tous les chemins de la scène, les points de l'espace sont les chemins, et la distribution de probabilité désirée pour les chemins est proportionnelle à leur contribution au rendu de l'image (la quantité d'illumination qu'ils apportent à la caméra). L'image finale est la projection de la distribution de ces chemins sur une image plane).

Pour éviter les pertes de performances dues à une acceptation des états trop faibles, on va accepter avec plus de facilité les chemins qui ont une quantité de lumière plus importante. Une fois les chemins sélectionnés on utilise une méthode de rendu de type path tracing bidirectionnel.

MCMC ou méthode classique

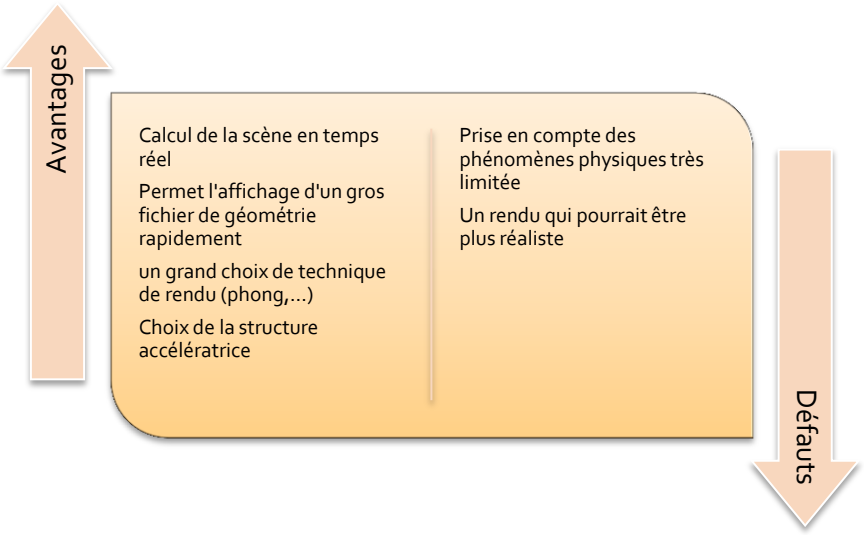
On a vu dans le IV. que, lors de notre rendu, on pouvait choisir entre la méthode MCMC ou classique. Si on sélectionne la méthode classique, le path tracing va parcourir chacun des chemins pour chaque rayon de la scène et à chaque rebond le rayon va être envoyé dans une direction aléatoire (principe de base du path tracing) pour ensuite récupérer la couleur de l'intersection avec une géométrie. Avant d'effectuer le rendu à proprement parler, la méthode MCMC a le grand avantage de calculer et trier les rayons qui contribuent le plus à la génération de l'image. Lorsque le rendu commence, le path tracing connaît déjà les chemins des rayons les plus intéressants à parcourir ce qui offre un rendu beaucoup plus rapide.

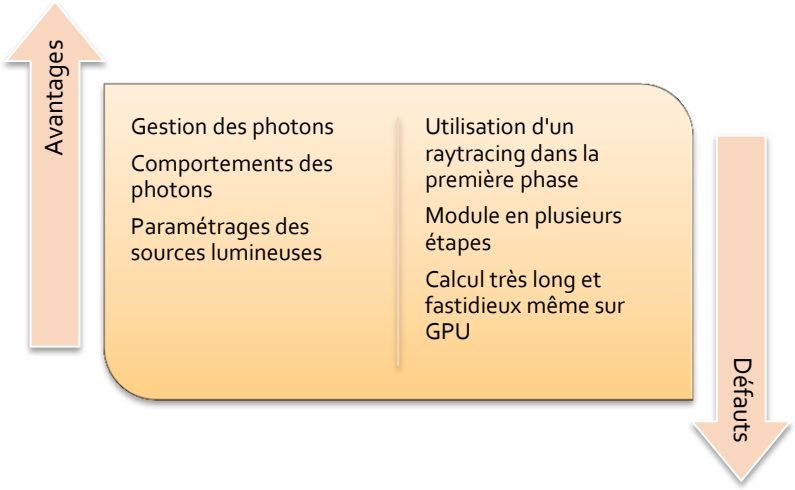
5. Comparatifs, avantages, défauts et rôles des différents modules

Nous avons donc un logiciel comprenant trois modules différents de rendu: un module de raytracing, un de photon mapping, un de path tracing. Nous allons essayer de voir dans cette partie les avantages et défauts de chacune de ces méthodes. Mais aussi, nous nous intéresserons aux rôles et dans quel cas les utiliser et enfin nous ferons un comparatif de rendu pour une même scène.

a. Avantages et défauts

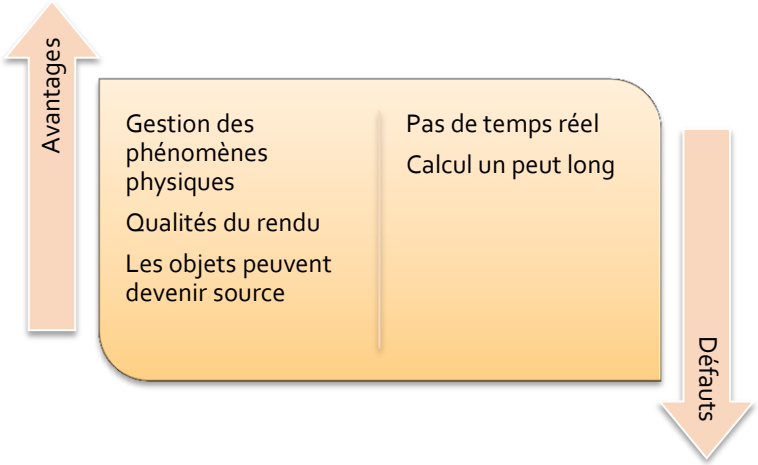
Commençons avec le raytracing qui est une technique de rendu 3d simpliste mais qui a l'avantage d'être en temps réel et donc qui permet de faire des visualisations rapides de la scène et de l'objet afin de préparer au mieux le rendu dans un autre mode qui sera lui beaucoup plus chronophage.
Je vois donc notre raytracer plus comme un outil de visualisation et de préparation pour les autres modules qui fourniront une image beaucoup plus réaliste


Avantages
Calcul de la scène en temps réel
Permet l'affichage d'un gros fichier de géométrie rapidement
un grand choix de technique de rendu (phong,...)
Choix de la structure accélératrice
Prise en compte des phénomènes physiques très limitée
Un rendu qui pourrait être plus réaliste
Défauts



Passons maintenant au module de photon mapping. Ce module est un module en plusieurs étapes qui permet d'avoir une gestion des photons et de la lumière beaucoup plus poussée. Mais ce module, de par son nombre d'étapes, souffre d'une lourdeur et d'une lenteur dans les calculs. Il servira surtout lorsque l'on veut voir avec précision le trajet et les interactions des photons avec l'objet 3d.

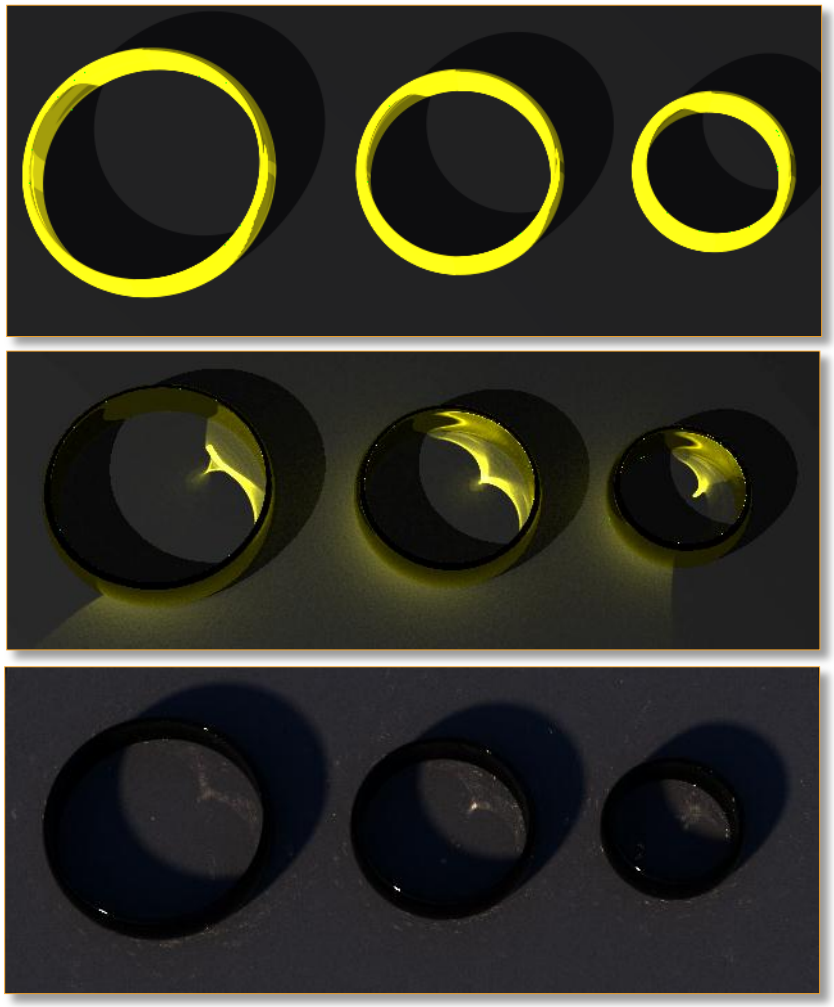
Pour finir, nous parlerons du path tracing qui est pour moi le module de rendu le plus précis et complet. On obtient grâce à lui, les meilleurs rendus d'objets malgré un temps de rendu assez long mais encore convenable. La prise en compte de nombreux effets physiques lui donne un grand potentiel.
Ce module servira donc à faire les rendus et les calculs sur lesquels l'on pourra ensuite travailler car il combine le meilleur des deux mondes.



b. Comparatifs des modules

Dans ce comparatif, je vais prendre trois scènes et je vais effectuer un rendu de ces scènes avec les trois modules afin que l'on puisse voir les différences entre chacun d'eux. Le temps de rendu est fixé à 30 secondes et les conditions de rendu seront les mêmes pour tous (même position des lumières,...).

Scène : ring.obj (plan diffus et anneau spéculaire)



Cette scène contient un plan diffus et des anneaux spéculaires. La lumière / soleil est au-dessus de la scène (pas complétement au zénith ce qui permet de faire apparaître les ombres et les caustiques).
Pour le raytracing, on a des réflexions et des ombres qui apparaissent mais on reste dans des affichages des effets très simples. Il n'y a aucune contribution de lumière ou de réflexion entre objets.
Pour le photon mapping, le rendu est très rapide et on remarque bien la qualité des caustiques et des reflexions. Enfin pour le path tracing, le temps de calcul n'est pas assez long, les effets de réflexions sont à peine visibles.

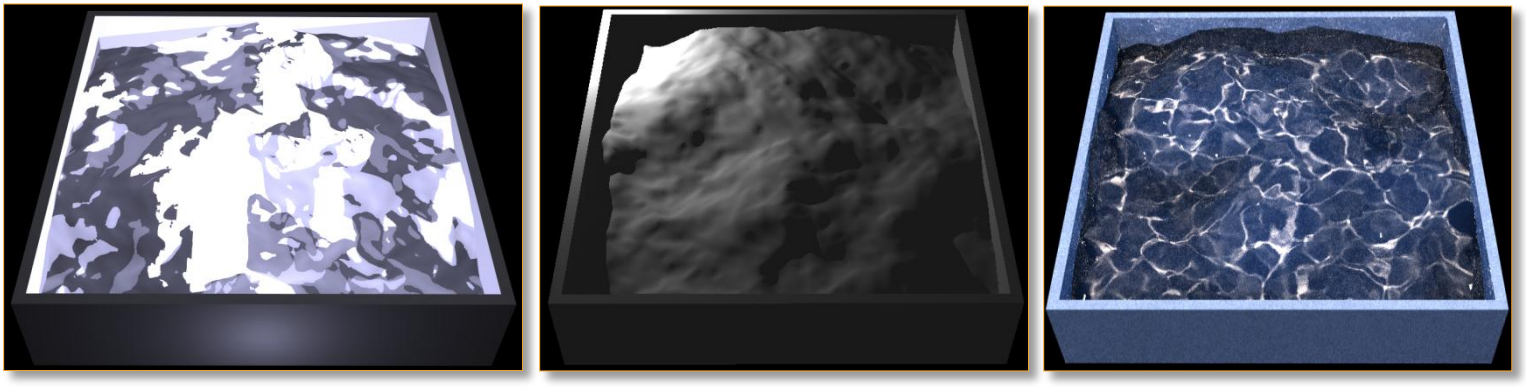
De haut en bas, rendu d'une scène avec le raytracing, photon mapping et path tracing.

Scène : altair.obj (objet avec textures)

Rendu de gauche à droite raytracing, photon mapping, et path tracing.

Cette scène contient un objet (un personnage) texturé avec des images sans matériaux réfléchissants. C'est clairement le raytracing qui prend l'avantage car il affiche instantanément un rendu quasi parfait. La scène étant un objet sans effet particulier la puissance du module lui permet de réussir sans problème. Pour le photon mapping, le rendu est basique (grâce à la pass de raytracing), l'éclairage qui devrait être beaucoup plus réaliste n'a pas le temps d'être calculé correctement en trente secondes. On perd donc le principal intérêt de cette méthode de rendu. Enfin, pour le path tracing, Il y a bien prise en compte des textures, cependant dans ce cas c'est le temps de calcul qui n'est pas assez long, on remarque du grain et des imperfections sur l'objet.

Scène : pool2.obj (« eau » complétement spéculaire)



Rendu de gauche à droite raytracing, photon mapping, et path tracing.

Pour cette dernière scène nous avons une boite diffuse et un plan déformé spéculaire. Pour le raytracing, les réflexions difficiles engendrées par le plan déformé spéculaire offre un rendu pas du tout satisfaisant. Le photon mapping se plante aussi car le temps de rendu est bien trop court pour qu'il puisse rendre la scène correctement à cause de toutes les réflexions possibles. Cependant le path tracing domine largement en termes de prise en compte des effets mais aussi en qualité de rendu.

Conclusion

Nous remarquons facilement que chaque module a son type de scène et que chacun doit rester dans ses champs de compétences si l'on veut garder des rendus corrects. Le raytracing sera donc plus performant sur les objets 3d sans effet et réflexion avec ou sans textures. Le photon mapping tirera son épingle du jeu pour des rendus d'objets avec des courbes où les caustiques sont nombreuses. Et enfin, le path tracing se révèlera redoutable pour les scènes faisant appel à de nombreux effets physiques et des réflexions très compliquées.

Ces tests bien qu'intéressant ne sont pas parfaits. En effet les trois modules sont programmés de manières bien différentes. Par exemple, pour les deux premiers modules, la lumière est une source ponctuelle qui ne faiblira pas avec la distance alors que pour le path tracing, nous avons un « soleil » qui prend en compte la distance avec l'objet et donc l'atténuation de l'illumination. Ce n'est qu'un exemple, mais il y a de nombreux petits détails qui empêchent de faire des tests identiques entre chaque module.

VI. Modèle thermique appliqué aux tuyères dans Crira

Comme nous l'avons vu précédemment, Crira est un logiciel qui a pour but principal d'établir la signature infrarouge (SIR) passive d'un aéronef en vol, en fonction de paramètres d'environnement globaux et propres à l'avion. Depuis le début du stage, je me suis surtout intéressé à la pass2 et à mon outil ARTGpu. Mais pour me donner du recul sur cet outil et pouvoir le finaliser, je vais m'intéresser pendant un mois à la pass1. Cette Pass1 calcule les paramètres généraux en fonction de la scène que l'on souhaite, et calcul le(les) jet(s). Pendant ce calcul des jets, elle calcule aussi la température des tuyères de l'appareil. C'est à ce moment que j'interviens. Le modèle thermique est très basique et doit être modifié. Je vais donc essayer de trouver les pistes et les améliorations possibles que l'on pourrait apporter.

1. Le modèle thermique actuel

A l'heure actuelle, la pass1 contient des modèles de turboréacteurs qui calculent les paramètres de sorties de turbines (température, vitesse,...). Ces modèles de turboréacteurs (*le fonctionnement des turboréacteurs sera expliqué plus loin*) sont les suivants :

- ⇨ Turboréacteur double flux double corps à flux mélangés.
- ⇨ Turboréacteur double flux simple corps à flux mélangés.
- ⇨ Turboréacteur double flux double corps à flux séparés.
- ⇨ Turboréacteur simple flux simple corps.

Cependant ces modèles, alors qu'ils calculent tous les paramètres en sortie de turbines, ne contiennent aucune fonction ou modèle pour le calcul de la température de la tuyère. Actuellement, la température de la tuyère est calculée par comparaison avec des mesures effectuées sur un Mirage III. On effectue donc une règle empirique pour connaître la température de la tuyère du modèle : $\alpha T_{turbine} + \beta T_{extérieur} = T_{tuyère}$. Mon rôle est donc de voir comment modifier et améliorer ce calcul afin de le rendre plus réaliste et précis.

2. Les turboréacteurs

Le **turboréacteur** est un système de propulsion essentiellement utilisé pour les avions. Un turboréacteur fonctionne sur le principe d'action-réaction. La variation de vitesse de l'air entre l'entrée et la sortie du réacteur crée une quantité de mouvements (dénommée poussée) vers l'arrière du moteur qui, par réaction, engendre le déplacement du moteur, donc du véhicule sur lequel il est fixé, vers l'avant.
Le turboréacteur fonctionne sur le principe des turbines à gaz. À l'admission, l'air est aspiré par la soufflante puis comprimé via un compresseur. Du kérosène est ensuite injecté puis mélangé avec l'air au niveau de la chambre de combustion puis enflammé, ce qui permet de fortement dilater les gaz. Ces derniers s'échappent du turboréacteur par la tuyère qui, en raison de sa section convergente, augmente la vitesse de l'air. L'air passe au préalable par une turbine permettant d'entraîner le compresseur et les accessoires nécessaires au fonctionnement du réacteur. Le mouvement est auto-entretenu tant qu'il y a injection de carburant. En simplifiant, l'énergie de pression engendrée au sein du réacteur sera transformée en énergie cinétique en sortie, ce qui engendrera une forte poussée (*voir schéma ci-dessous*).

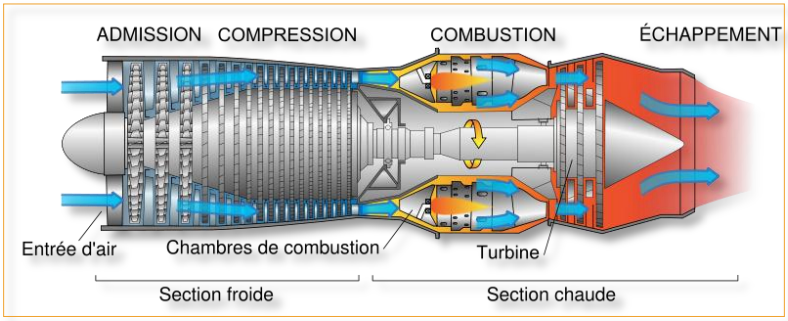


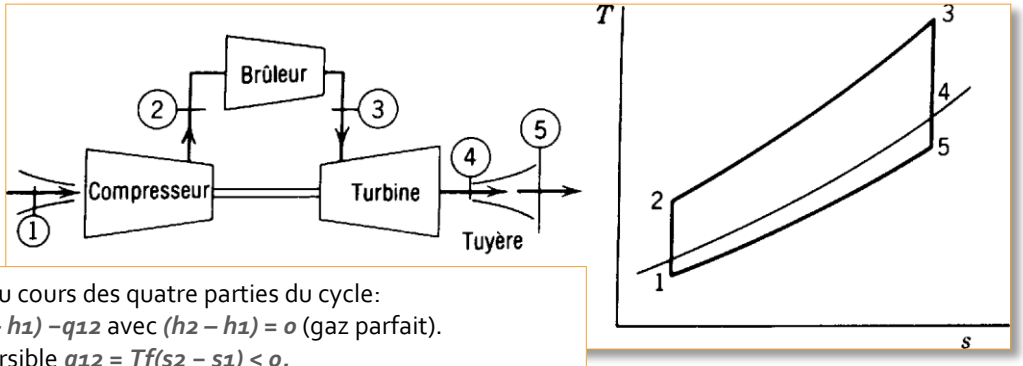
Schéma du fonctionnement d'un turboréacteur.

Le turboréacteur réalise ainsi un cycle continu à quatre temps — admission, compression, combustion et détente/échappement, théoriquement décrit par le cycle de Brayton.

Explication sur le cycle de Brayton ou cycle de Joule (issue d'un livre de thermodynamique)

Le cycle de Joule se compose des mêmes transformations que le cycle de Rankine (2 échanges de chaleur isobares et deux variations de pression isentropiques), la seule différence étant que le fluide actif reste toujours à l'état gazeux. C'est le cycle idéal de la turbine à gaz. Le cycle du turboréacteur est également une variante du cycle de Joule, dans laquelle on effectue une détente partielle dans la turbine, de manière à ce que la puissance fournie par la turbine soit juste suffisante à entraîner le compresseur. Les gaz à la sortie de la turbine sont alors détendus dans une tuyère pour être accélérés et ainsi produire une poussée.

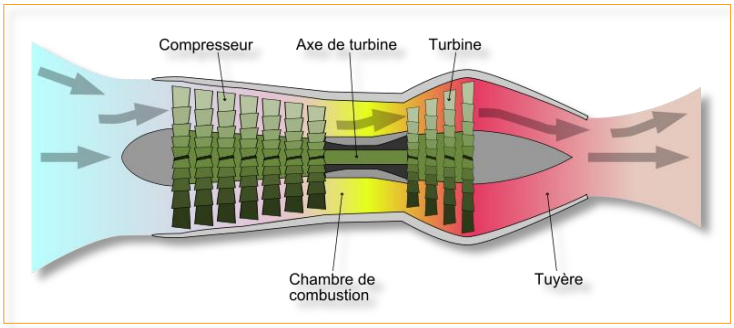
Cycle de Brayton pour un turboréacteur (source cours thermodynamique).



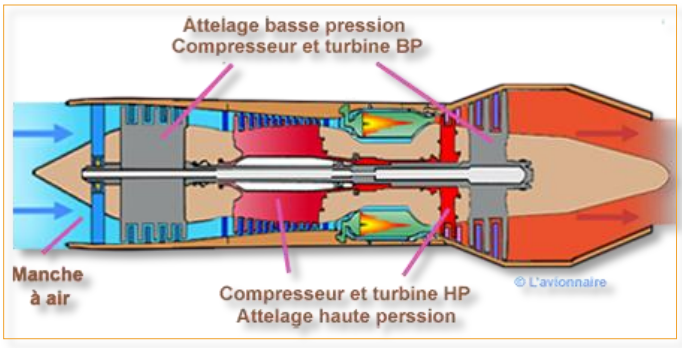
Quantités de chaleur et travail échangés au cours des quatre parties du cycle:
Compression isotherme 1 – 2 $W_{12} = (h_2 - h_1) - q_{12}$ avec $(h_2 - h_1) = 0$ (gaz parfait).
Et en supposant l'échange de chaleur réversible $q_{12} = T_f(s_2 - s_1) < 0$.
Chauffage isobare 2 – 3 $q_{23} = h_3 - h_2$
Détente isotherme 3 – 4 $W_{34} = (h_4 - h_3) - q_{34}$ avec $(h_4 - h_3)=0$ (gaz parfait).
Et en supposant l'échange de chaleur réversible $q_{34} = T_c(s_4 - s_3) > 0$.
Refroidissement isobare 4 – 1 $q_{41} = h_1 - h_4$. On a donc $q_{23} = -q_{41}$.

Les différents types de turboréacteurs (définitions issues de wikipédia et l'avionnaire)

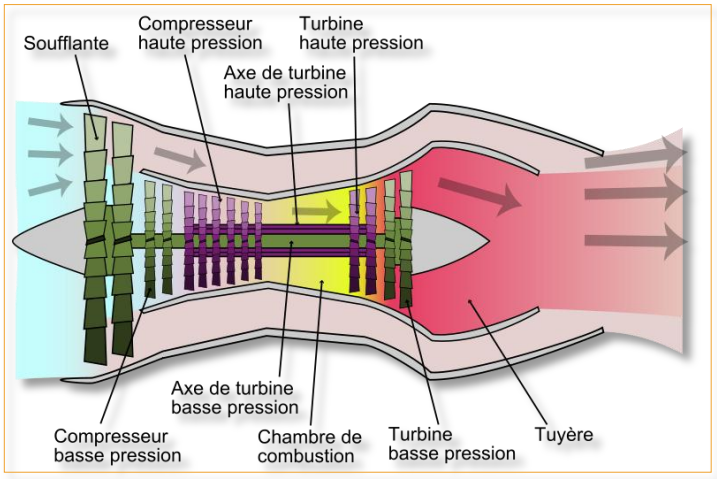
Simple et multi corps. On appelle **corps** un ensemble compresseur-turbine accouplés sur un même arbre et tournant donc à la même vitesse. Un turboréacteur peut être mono, double ou triple corps. Dans les premiers turboréacteurs, turbine et compresseur forment un unique ensemble cinématique. On parle alors de compresseurs / moteurs **simple corps** ou simple attelage *(image ci-contre)*.



Pour accroître l'efficacité du turboréacteur, le compresseur est désormais divisé en deux parties successives : une à basse pression et une à haute pression, mues par deux turbines successives (haute et basse pression). On parle alors de turboréacteurs **double corps** ou double attelage (*voir image ci-dessous*).



Simple ou double flux. Les moteurs sont dits à **simple flux** quand la totalité de l'air admis traverse le moteur et actionne les turbines. Les turboréacteurs à simple flux sont très bruyants et n'atteignent leur meilleur rendement qu'au-delà de Mach 1 (*exemple de turboréacteur simple flux simple corps image précédente*).



Bien plus économiques aux vitesses subsoniques et moins bruyants, les turboréacteurs à **double flux** (*image ci-contre*) sont apparus dans les années 1960. Dans ces moteurs, les premiers étages du compresseur basse pression, souvent réduits à la soufflante, sont de grandes dimensions pour aspirer de grandes quantités d'air. L'air pré-comprimé par la soufflante ne passe pas intégralement par le moteur, mais une partie (le flux froid) le contourne par sa périphérie jusqu'à la tuyère où il est éjecté avec les gaz chauds (flux chaud). De plus, selon le type de turboréacteurs, les deux flux (froid et chaud) éjectés vont être soit mélangés soit séparés. Ce second flux n'est pas mélangé mais vient "gainer" le flux primaire fortement accéléré. Cela permet, pour des vitesses modérées, en dessous de Mach 1,5 environ, d'augmenter la poussée par augmentation du débit de gaz, et de réduire considérablement le niveau de bruit. La proportion d'air contournant le moteur est variable selon les moteurs. Elle est d'autant plus élevée que le moteur est destiné à voler à des vitesses faibles.

La tuyère

Nous pourrions passer beaucoup de temps à parler de chaque composant d'un turboréacteur. Bien que très intéressant, je ne parlerais que de la tuyère sur laquelle porte ce sujet.

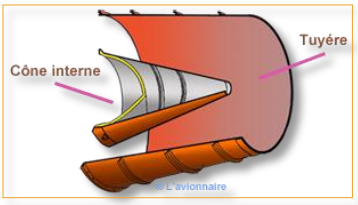
Tuyères de turboréacteurs (source L'avionnaire).

Le rôle de la tuyère est d'évacuer les gaz chauds sous pression sortant des turbines. Elle va poursuivre la détente des gaz sortant de la turbine et donc transformer l'énergie potentielle en énergie cinétique. Cette transformation procure une poussée (le reste de la poussée provenant du moteur et de la prise d'air). L'arrière-corps est la partie externe de la tuyère. Pour les avions subsoniques, les tuyères sont convergentes ; les flux primaire et secondaire peuvent être séparés, confluents ou mélangés.

Pour les avions supersoniques, les tuyères sont convergentes divergentes. Les sections du col et de sortie sont réglables de manière à assurer un bon fonctionnement de la tuyère dans tout le domaine de vol (subsonique, supersonique avec et sans réchauffe, *image ci-contre dessus*).

L'image ci-contre présente une tuyère de turboréacteur d'un avion subsonique à double flux séparés. En bleu la tuyère du flux secondaire et en rouge la tuyère du flux primaire.

3. Etude thermodynamique de la tuyère



Dans cette partie, je vais présenter les différents phénomènes qui se produisent dans une tuyère. Nous faisons bien sûr l'approximation que les gaz sont parfaits et que l'avion reste à une altitude constante pendant le calcul. Dans un premier temps, je calculerai les caractéristiques du gaz et de la tuyère (température, pression, col, …). Dans un second temps, je montrerai les formules pour le mélange des gaz (cas des turboréacteurs doubles flux internes). Et enfin, je mettrai en place le calcul des températures du matériau composant la tuyère.

a. Calcul des principaux paramètres d'une tuyère convergente

Vitesse de son dans l'air : On montre en mécanique que le son est une onde de pression dont la vitesse de propagation est $a = \frac{1}{\sqrt{\rho\,\chi s}}$ avec ρ la masse volumique et χs le coefficient de compressibilité adiabatique.

$\chi s = -\frac{1}{v}.(\frac{dv}{dp})_s$ $\chi t = -\frac{1}{v}.(\frac{dv}{dp})_t$ v est le volume massique.

Grâce à la relation de Reech entre la pente de l'adiabatique $p\,.\,v^{\gamma} = cte$ et la pente de l'isotherme $p\,.\,v = cte$.

$(\frac{dv}{dp})_s = \frac{1}{\gamma}(\frac{dv}{dp})_t$ d'où $\chi t = \gamma\,.\chi s$ ce qui donne $a = \sqrt{\frac{\gamma}{\rho\,\chi t}}$.

Pour un gaz parfait $\chi t = \frac{1}{p}$ d'où $a = \sqrt{\frac{\gamma p}{\rho}} = \sqrt{\gamma p v}$.

Ecoulement adiabatique sans frottement dans un conduit horizontal : on a un débit massique constant avec S la section du conduit et c la vitesse du fluide :

$$Dm = \frac{dm}{dt} = \frac{d(\rho\,.\,v)}{dt} = \rho S\,.\frac{dx}{dt} = \rho S c = \frac{S\,c}{v}$$

Le fait que l'écoulement soit stationnaire, donc que le débit massique soit constant, se traduit par : $\frac{dS}{S} = \frac{dv}{v} - \frac{dc}{c}$

L'équation pour un écoulement dans une turbine idéale est $v\,.\,dp = -c\,.\,dc$.Cette équation montre bien que pour augmenter la vitesse d'écoulement il faut diminuer la pression dans le fluide. L'écoulement est adiabatique réversible, on a donc pour un gaz parfait $\frac{dp}{p} + \gamma\,.\frac{dv}{v} = 0$.

Et en utilisant la loi de l'écoulement $v\,.\,dp = -c\,.\,dc$ on a $\frac{dv}{v} = \frac{1}{\rho\,p}\,.\frac{c\,dc}{v}$ puis en reportant dans la loi du débit massique on obtient :

$$\frac{dS}{s} = \frac{1}{\gamma p v}\,.\,c\,.\,dc - \frac{dc}{c} = \left[\frac{c^2}{\gamma p v} - 1\right].\frac{dc}{c}$$

On appelle le nombre de Mach (Ma) le rapport entre la vitesse du fluide et la célérité du son $Ma = \frac{c}{a} = \frac{c}{\sqrt{\gamma p v}}$.

D'où l'équation $\frac{dS}{s} = [Ma^2 - 1]\,.\frac{dc}{c}$.

On peut aussi relier la variation de section à l'écoulement en divisant l'équation du fluide en écoulement par c² et on a:

$$\frac{dc}{c} = -\frac{1}{c^2}\,.vdp = -\frac{va^2}{c^2\gamma p v}dp = -\frac{1}{\gamma Ma^2}\,.\frac{dp}{p} \text{ soit } \frac{dS}{S} = \frac{1 - Ma^2}{\gamma\,.\,Ma^2}\,.\frac{dp}{p}$$

Ces deux équations mettent en évidence des comportements différents suivant que le nombre de Mach est plus petit ou plus grand que 1.

Pour M<1, la vitesse du fluide est inférieure à celle du son, on a donc un écoulement dit subsonique. Pour que la vitesse du fluide augmente (dc>0) il faut que la section du conduit diminue (dS<0), on a donc la pression du fluide qui diminue (dp<0).

Pour M>1, on a un écoulement supersonique. Pour que la vitesse du fluide augmente il faut ***dc>o*** donc on a ***dS>o*** la section du conduit augmente, par contre on a toujours ***dp<o*** la pression du fluide diminue. Cette situation est inconcevable pour un fluide incompressible mais ici on a :

$$\frac{dv}{v} = \frac{1}{\gamma p} \cdot \frac{cdc}{v} = \frac{c^2}{\gamma pv} \cdot \frac{dc}{c} = Ma^2 \frac{dc}{c} \text{ donc } \frac{dv}{v} > \frac{dc}{c}$$

En régime supersonique la variation relative du volume massique est plus grande que celle de la vitesse, c'est l'expansion volumique du fluide qui impose l'augmentation de la section du conduit.

Pour M=1 on a $\frac{1}{S} \cdot \frac{dS}{dx} = \frac{Ma^2 - 1}{c} \cdot \frac{dc}{dx} = 0$ donc il s'agit d'un minimum de section du canal : le col. Pour dépasser la vitesse du son en sortie il faut donc une tuyère convergente-divergente La section du canal diminue pour augmenter la vitesse du fluide jusqu'à ce qu'elle atteigne la vitesse du son puis elle ré-augmente ensuite. La tuyère doit avoir un profil calculé très précisément pour obtenir cet effet.

Calcul des valeurs au col : On prend pour indice ***e*** pour l'entrée de la tuyère et ***s*** pour la sortie au col. Au col on a $cs^2 = as^2 = \gamma \cdot ps \cdot vs$.

$$vs = ve \cdot \left(\frac{pe}{ps}\right)^{\frac{1}{\gamma}}$$

$$ps \cdot vs = ps \cdot ve \cdot \left(\frac{pe}{ps}\right)^{\frac{1}{\gamma}} = pe \cdot ve \cdot \left(\frac{ps}{pe}\right)^{\frac{\gamma-1}{\gamma}}$$

$$cs^2 = \gamma \cdot pe \cdot ve \cdot \left(\frac{ps}{pe}\right)^{\frac{\gamma-1}{\gamma}}$$

En intégrant la loi de l'écoulement $v \cdot dp + c \cdot dc = 0$ il vient :

$$\frac{2\,cs}{2} - \frac{2\,ce}{2} + \frac{\gamma}{\gamma - 1} \cdot (ps \cdot vs - pe \cdot ve) = 0$$

$$\left(\frac{ps}{pe}\right)^{\frac{\gamma-1}{\gamma}} = 1 - \frac{cs^2 - ce^2}{2 \cdot pe \cdot ve} \cdot \frac{\gamma - 1}{\gamma} = 1 + \frac{ce^2}{\gamma \cdot pe \cdot ve} \cdot \frac{\gamma - 1}{2} - \frac{cs^2}{\gamma \cdot pe \cdot ve} \cdot \frac{\gamma - 1}{2}$$

$$\left(\frac{ps}{pe}\right)^{\frac{\gamma-1}{\gamma}} = 1 + \frac{ce^2}{ae^2} \cdot \frac{\gamma - 1}{2} - \left(\frac{ps}{pe}\right)^{\frac{\gamma-1}{\gamma}} \cdot \frac{\gamma - 1}{2}$$

$$\left(\frac{ps}{pe}\right)^{\frac{\gamma-1}{\gamma}} = \frac{2}{\gamma + 1} + \frac{\gamma - 1}{\gamma + 1} \cdot \frac{ce^2}{ae^2}$$

$$\left(\frac{ps}{pe}\right) = \left(\frac{2}{\gamma + 1} + \frac{\gamma - 1}{\gamma + 1} \cdot Mae^2\right)^{\frac{\gamma}{\gamma-1}}$$

On a donc les résultats suivant avec lesquels on peut calculer toutes les caractéristiques du gaz/fluide et de la tuyère (avec ***s*** la sortie et ***e*** l'entrée pour la tuyère) :

Pression	$\left(\frac{ps}{pe}\right) = \left(\frac{2}{\gamma + 1} + \frac{\gamma - 1}{\gamma + 1} \cdot Mae^2\right)^{\frac{\gamma}{\gamma-1}}$
Température gaz / fluide	$\left(\frac{Ts}{Te}\right) = \left(\frac{2}{\gamma + 1} + \frac{\gamma - 1}{\gamma + 1} \cdot Mae^2\right)$
Volume	$\left(\frac{vs}{ve}\right) = \left(\frac{2}{\gamma + 1} + \frac{\gamma - 1}{\gamma + 1} \cdot Mae^2\right)^{\frac{1}{1-\gamma}}$
Vitesse d'écoulement au col	$cs = \sqrt{\gamma \cdot ps \cdot vs} = \sqrt{\gamma \cdot pe \cdot ve \cdot \left(\frac{2}{\gamma + 1} + \frac{\gamma - 1}{\gamma + 1} \cdot Mae^2\right)}$
La section	$Dm = \frac{Se \cdot ce}{ve} = \frac{Ss \cdot cs}{vs}$ $\frac{Ss}{Se} = \frac{vs}{ve} \cdot \frac{ce}{cs} = Mae \cdot \left(\frac{2}{\gamma + 1} + \frac{\gamma - 1}{\gamma + 1} \cdot Mae^2\right)^{\frac{\gamma+1}{2(1-\gamma)}}$

b. Cas des turboréacteurs double flux

Nous venons de voir comment calculer la pression, température du gaz / fluide, volume, section, vitesse d'écoulement pour une tuyère convergente. Ces calculs peuvent être utilisés dans n'importe quel cas avec n'importe quel turboréacteur car les calculs permettent de trouver les valeurs en sorties si on connaît les valeurs d'entrées. Cependant, si on utilise un turboréacteur double flux internes (le mélange des gaz est fait avant la tuyère et non pas éjecté par deux endroits différents), on doit calculer la température du mélange des deux gaz (Crira ne le fait pas) afin d'avoir des valeurs justes lors du calcul des valeurs de la tuyère.

Nous avons donc deux gaz parfaits qui vont être mélangés. Ces gaz étant considérés comme parfaits. La variation globale d'énergie interne est nulle lorsque nous les mélangeons. On peut donc utiliser la formule suivante :

$$n1 \,.\, Cv1 \,.\, (T - T1) + n2 \,.\, Cv2 \,.\, (T - T2) = 0$$

Ce qui nous permet de trouver la température résultante des gaz / fluides :

$$T = \frac{n1 \,.\, Cv1 \,.\, T1 + n2 \,.\, Cv2 \,.\, T2}{n1 \,.\, Cv1 + n2 \,.\, Cv2}$$

Mais aussi la pression :

$$\frac{pV}{T} = p1 \,.\, \frac{V1}{T1} + p2 \,.\, \frac{V2}{T2}$$

$$p = \frac{T}{V}\left(p1 \,.\, \frac{V1}{T1} + p2 \,.\, \frac{V2}{T2}\right)$$

Nous pouvons donc, maintenant que l'on connaît la température résultante des deux gaz et la pression, utiliser les formules du a. pour calculer les valeurs de la pression, température, et autres dans la tuyère.

c. Température de la tuyère

Pour finir, il faut connaître la température de la tuyère que le gaz va échauffer ou refroidir. Toujours dans le même esprit que précédemment, on reste dans l'hypothèse que l'on est en présence de gaz parfait. On va rajouter une approximation en plus : on a conservation de la chaleur (on n'a pas de perte, la chaleur fournit par le gaz est entièrement récupéré par la tuyère).

$$dQ = m \,.\, cv \,.\, dTd$$

Avec Q la chaleur, m la masse, cv la chaleur massique à volume constant et dTd la différence de température entre l'entrée de la tuyère et la sortie pour le gaz / fluide.

Comme expliqué tout à l'heure, on dit que l'on a conservation de la chaleur donc $dQ^{\swarrow} = dQ^{\nearrow}$. Dans un métal, les atomes sont disposés selon un réseau cristallin à trois dimensions. Chaque atome du réseau peut vibrer selon trois directions perpendiculaires entre elles, qui ont chacune deux degrés de liberté.

En admettant que le métal ne travaille pas (dilatation négligeable), toute quantité de chaleur dQ échangée avec le milieu ambiant sert à modifier l'énergie interne du métal de la quantité dU et, ainsi, à varier sa température de dT (M = masse atomique du métal [kg·kmol-1]) :

$$dQ = dU = 3 \cdot \frac{m}{M} \cdot R \cdot dT$$

La chaleur massique d'une substance, notée c en [J·kg-1·K-1], exprimant l'énergie nécessaire pour faire varier de 1°C (ou 1K) la température d'une masse de 1 kg de cette substance, on écrit :

$$dQ = c \cdot m \cdot dT \quad \text{avec} \quad c = \frac{3\,R}{M}$$ Loi de Dulong-Petit

On a donc pour la formule pour connaître l'élévation de la température de la tuyère :

$$dT = \frac{dQ}{c \cdot m} \qquad \frac{dQ \,.\, M}{3.\ m \,.\, R} = \frac{dU \,.\, M}{3.\ m \,.\, R} = dT$$

d. Programmation du modèle

J'ai donc développé un petit programme en C++ qui va permettre de calculer la température de la tuyère à partir des paramètres d'environnement (pression atmosphérique, température de l'air, masse volumique de l'air, valeur Mach,...) mais aussi grâce aux valeurs en sortie de turbine des turboréacteurs (température, pression et vitesse du fluide). Ce petit programme est composé de deux fichiers cpp (l'un contenant les fonctions de calculs, l'autre contenant le main) et un fichier .h (contenant les includes et les prototypes de fonctions).

Il fonctionne de la manière suivante : on entre en premier les paramètres d'environnement (pression et température) puis les paramètres de la tuyère (diamètre d'entré, diamètre de sortie, longueur) et enfin les paramètres en sortie de turbine (température, pression et vitesse du fluide). Ensuite le programme va calculer la masse volumique de l'air, la vitesse du son et la valeur du nombre Mach. Une fois ces valeurs connues, on calcul la valeur de la pression de sortie afin de pouvoir par la suite calculer la température de sortie de tuyère et ainsi connaître la différence de température du fluide entre l'entrée de la tuyère et la sortie.
Cette différence de température va permettre de connaître la quantité de chaleur fournie à la tuyère elle-même. Il faut bien sûr au préalable calculer la masse volumique d'air dans la tuyère, le volume de la tuyère et la masse d'air dans ce volume. On peut enfin calculer la température de la tuyère.

Il est aussi implémenter une partie de code pour le refroidissement de la tuyère par l'air extérieur.

Problème

A l'heure où j'écris ce paragraphe du rapport, le logiciel ne fournit pas les valeurs finales cohérentes pour la température de la tuyère. Je suis sûr que jusqu'au calcul de l'échauffement tout est juste (vérification avec des modèles trouvés sur internet et dans des cours pour la masse volumique de l'air, le nombre Mach, pression et température du fluide).
Ces valeurs incomplètes viennent du fait que l'on calcul la température d'échauffement sur un temps dt pour un seul volume de tuyère. Cependant le fluide à une vitesse, pour avoir l'échauffement total de la tuyère pendant le fonctionnement de l'appareil, il faut faire en sorte que le programme prenne en compte le temps total pendant lequel l'air chaud est éjecté.
Il faut donc ajouter cette notion de vitesse de fluide afin d'avoir la valeur finale de la température de la tuyère. Connaissant la vitesse du fluide, le volume de la tuyère, l'élévation de la température sur un temps dt, il sera facile de mettre en place ce calcul. De plus, le refroidissement par l'air extérieur sera aussi pris en compte. La tuyère devrait donc chauffer jusqu'à une valeur critique pour ensuite se stabiliser.

VII. Conclusion

L'objectif premier de ce stage était la réalisation d'une subdivision de l'espace sous GPU. Comme on l'a vu tout au long de ce rapport, j'ai développé un logiciel outil ARTGpu. Ce logiciel est une vitrine des possibilités qu'offre Optix. Avec cet outil, j'ai essayé d'intégrer le plus grand nombre de méthodes de rendu, d'options de rendu et de fonctions qui pourraient, dans le futur, intéresser pour le logiciel Crira. Il permet aussi de faire de nombreux tests sur les différents types de rendu afin de choisir au mieux les technologies et techniques à utiliser. Cependant l'intégration directe de cet outil dans Crira est compromise à cause de l'impossibilité du langage assembleur ptx (utilisé par Optix) à écrire dans un fichier les résultats nécessaires au bon fonctionnement de Crira. Il faut donc trouver une solution à ce problème ou alors intégrer directement un module de rendu en infrarouge dans ARTGpu.
J'ai aussi pu m'intéresser aux tuyères et aux turboréacteurs en commençant à programmer un petit modèle physique qui va calculer les températures des tuyères durant le fonctionnement d'un appareil d'une façon un peu plus précise que ce qui est fait à l'heure actuelle.

Durant ces quelques mois j'ai eu la chance/satisfaction de travailler dans des domaines qui me passionnent comme le rendu d'image par ordinateur, la programmation et l'aéronautique. Mais je n'ai pas fait que programmer, je me suis intéressé aussi à l'aspect physique qui se cache derrière et c'est en cela que ce fut réellement intéressant. Il a fallu comprendre comment ces phénomènes fonctionnent pour pouvoir les développer correctement et les faire fonctionner.

Bien que ce stage fût plus tourné vers l'informatique, il m'a conforté dans le potentiel du Master Physique et Informatique qui m'a rendu indépendant en informatique tout en me permettant de comprendre la physique et avoir la réflexion et les méthodes nécessaires à la compréhension des phénomènes qui nous entourent.
J'ai également eu la chance d'être autonome dans mon travail, de gérer le projet, de suivre un planning, de prendre les décisions nécessaires et de proposer des solutions quand un problème survient. Mais ce stage m'a aussi permis de travailler en relation avec des gens, d'écouter leurs demandes et leurs besoins, de trouver des possibilités à leurs problèmes et de rechercher des solutions en groupes. Bien qu'indépendant, le travail de groupe a été nécessaire et extrêmement bénéfique.

D'un point de vue plus technique, j'ai pu m'améliorer et découvrir de nombreux langages de programmation (perl, c++, cuda, optix, html5, css3,...), me remettre à la thermodynamique et à la physique de rendu.

Cette expérience aussi bien humaine que professionnelle va me permettre de commencer ma vie future dans les meilleures conditions possibles.

VIII. Bibliographie / Webographie

Partie I : Présentation

"Qui sommes-nous ?"
www.onera.fr

Documents internes à l'Onera (intranet, bibliothèque)

Partie II : Introduction aux notions

"Introduction aux Octrees "
" Développement d'un raytracer"
www.algr.org/docs/

Le photon mapping
http://www.mti.epita.fr/blogs/2012/05/01/le-photon-mapping/

CUDA / OPTIX
www.nvidia.com (et tous ses sous domaines, forums,…)

"CUDA par l'exemple"
Ce livre est une introduction à la programmation parallèle sous GPU écrit par Jason Sanders et Edward Kandrot (tous deux ingénieurs chez Nvidia) pour les éditions Pearson

"Cours GPU"
Document interne à l'Onera, il est une introduction à la programmation sous CUDA

"Cours CUDA"
Introduction au calcul sous CUDA
http://libpfb.so/
P-F Bonnefoi

"Le kd-Tree : une méthode de subdivision spatiale"
Université de Rennes 1
Cédric Fleury

"Fast, Minimum Storage Ray / Triangle Intersection"
Université de Virginie
http://www.cs.virginia.edu/~gfx/
Tomas Moller et Ben Trumbore

"Manifold Exploration: A Markov Chain Monte Carlo technique for rendering scenes…"
Université de Cornell
Wenzel Jakob et Steve Marschner

"Stackless KD-Tree Traversal for High Performance GPU Ray Tracing"
Eurographics 2007, Volume 26, Number 3
Stefan Popov, Johannes Gunther, Hans-Peter Seidel, Philipp Slusallek

"Real-Time KD-Tree Construction on Graphics Hardware"
www.kunzhou.net
Kun Zhou, Qiming Hou, Rui Wang, Baining Guo

"Understanding the Efficiency of Ray Traversal on GPUs"
http://code.google.com/p/understanding-the-efficiency-of-ray-traversal-on-gpus/
Timo Aila, Samuli Laine

Logiciel 3D : blender
www.blender.org

Partie III : Problématiques et solutions

Rapport de stage de Charles Thierry

"A Performance Comparison of CUDA and OpenCL"
Kamran Karimi, Neil G. Dickson, Firas Hamze

Partie IV : Développement et programmation

Langage de programmation (perl, c++, cuda, optix)
www.lesiteduzero.com
www.developpez.org
http://formation-perl.fr/
www.nvidia.com (et tous ses sous domaines, forums,...)

Banque d'objet obj et utilisation de Nview
http://artist-3d.com/
www.xnview.com

Partie V : Les méthodes de rendu

Comprendre les différentes techniques de rendu graphique
http://web.cs.wpi.edu
http://graphics.ucsd.edu/~iman/BDPT/
http://biblio.alloprof.qc.ca

"Ray Tracing on GPU"
Martin Christen

"Ray Tracing on GPU"
Nitish Jain

"Ray Tracing"
Irisa
Kadi Bouatouch

"... A Markov Chain Monte Carlo technique for rendering scenes with difficult specular transport"
Université de Cornell
Wenzel Jacob, Steve Marschner

Partie VI : Modèle thermique

Définition turboréacteur et tuyère
http://lavionnaire.fr

"Cours de Thermodynamique"
Olivier Cleynen

"Cours d'Aérothermodynamique"
Dr Henni Mansour Z

"Thermodynamique technique"
J.F. Devois

"Fluides Compressibles"
Pascal Bigot

ANNEXES

Fiche de programmation Optix et Cuda
www.nvidia.com (et tous ses sous domaines, forums,...)

De nombreux ouvrages papiers ont aussi été utilisé comme les livres de physiques aux éditions Dunod et informatiques aux éditions Person mais cela prendrais trop de place à tous les citer.

IX. ANNEXES

1. La programmation avec Cuda

Il sera traité dans cette annexe les définitions et les notions essentielles de programmation. Pour tout ce qui est "comment fonctionne CUDA?" voir dans le rapport de stage partie II.

a. Introduction

Cuda est un environnement logiciel qui permet d'utiliser le GPU,"Graphics Processing Unit" au travers de programme de haut niveau comme le C et le C++. Ainsi le programmeur écrit un programme C avec des extensions CUDA, de la même manière qu'un programme OpenMP. La seule obligation est d'avoir une carte graphique Nvidia ayant une architecture Fermi ou Kepler. Les fichiers doivent être compilés avec le compilateur Nvidia NVCC.

Définitions

MIMD: Partitionnement global en tâches de calcul dépendantes ou non: *Gros Grain*.
SIMD: Partitionnement local, on découpe en sous domaines, en blocs. On a des opérations primitives identiques sur des données différentes: *Grain Fin.*
Host ou CPU: c'est l'ordinateur qui sert d'interface avec l'utilisateur et qui contrôle le **device** utilisé pour exécuter les parties de calculs intensifs basés sur un parallélisme de données. Il peut **R / W** dans Global memory, constant memory et **R** dans texture memory. *L'hôte est responsable de l'exécution des parties séquentielles de l'application.*
GPU: est le processeur graphique, "General-Purpose Graphics Processor Unit".
Device: est le GPU connecté à **l'hôte** et qui va exécuter les parties de calcul intensif basé sur un parallélisme de données. *Le device, ou périphérique, est responsable de l'exécution de la partie parallèle de l'application.*
Kernel: est une fonction qui peut être appelée depuis **l'hôte** et qui est exécutée en parallèle sur le **device** CUDA par de nombreux **threads**. Le **kernel** est exécuté simultanément par des milliers de **threads**.

b. Notions de threads, grille et blocs

CUDA adopte la hiérarchie suivante: un **kernel** est un programme exécuté par un ensemble de **threads** indépendants s'exécutant en parallèle sur le GPU.
Les **threads** (fils d'instructions) sont groupés en **blocs** qui coopèrent et traitent le même code écrit en CUDA/C sur les données différentes liées au numéro de chaque **thread** (Max 1024 threads par bloc).
Les **blocs** sont groupés en **grille** et s'éxécutent dans n'importe quel ordre, il n'y a pas de synchronisation inter-blocs. (Max Dg.x=< 65535, Dg.y=<65535, Dg.z = 1). Les **threads** de 2 blocs différents ne communiquent pas.

Les appels et identifications des blocs / threads

Nombre de blocs dans la grille: dim3 **gridDim** (gridDim.x , grimDim.y , grimDim.z=1)
Nombre de threads par blocs: dim3 **blockDim** (blockDim.x , blockDim.y , blockDim.z)
Indice du bloc dans la grille: dim3 **blockIdx** (blockIdx.x , blockIdx.y , blockIdx.z=1)
Indice du thread dans le bloc: dim3 **threadIdx** (threadIdx.x , threadIdx.y , threadIdx.z)
Taille du WARP en nombre de threads: int **warpSize**

Un warp en CUDA est donc un ensemble de 32 threads, il s'agit de la taille minimale des données traitées de façon SIMD par un multiprocesseur en CUDA.

Bloc 1D: **ThreadID** = threadIdx.x = x
Bloc 2D: **ThreadID** = threadIdx.x + blockIdx.y * blockDim.x
Bloc 3D: (Dx , Dy , Dz) = (blockDim.x , blockDim.y , blockDim.z)
ThreadID = threadIdx.x + threadIdx.y * Dx + threadIdx.z * Dx * Dy

Exemple: kernel<<<nBlocks, nThreads>>>();
int index_x = threadIdx.x + blockIdx.x * blockDim.x
int index_y = threadIdx.y + blockIdx.y * blockDim.y
Les threads peuvent: **R / W** dans registers, shared memory, local memory, global memory.
R dans constant memory, texture memory.

Bien dimensionner ses blocks et sa grille

Un point important pour obtenir de bonnes performances : le dimensionnement des *blocks* et de la grille.
Nous l'avons vu ci-dessus : un *block* s'exécute sur un unique multiprocesseur. Donc, si une carte graphique possède *n* multiprocesseurs, il faut bien sûr que la grille contienne au moins *n blocks* pour que chaque multiprocesseur soit actif.
Nous savons aussi que les threads d'un *block* sont exécutés par *warp*, c'est à dire par paquet de 32. Il est donc idéal d'avoir un nombre de threads par *block* qui soit un multiple de 32, pour éviter de se retrouver avec des *warps* dont des threads ne font rien.
De plus, il est important d'avoir beaucoup de threads dans un *block*, de manière à masquer les temps de latence des opérations mémoires et calculs. En effet, lorsqu'une opération est effectuée par un thread, son résultat n'est pas disponible immédiatement. Il se passe de quelques dizaines à quelques centaines de cycles d'horloge avant qu'il soit disponible. Pendant ce temps-là, si la suite de l'exécution du thread dépend de cette valeur, le thread est bloqué. C'est pourquoi il faut que le multiprocesseur ait d'autres threads disponibles à exécuter pendant ce temps de latence. C'est ce qu'on appelle **masquer la latence**.
Pour autant, il ne suffit pas de donner une grande taille à un *block* pour avoir de bonnes performances, et ce pour deux raisons :

- ⇨ Une carte de génération 1.3 ne peut avoir que 1024 threads résident. Si un *block* contient 768 threads, alors il est impossible de charger un autre *block* et seuls 768 threads seront résidents là où avec un *bloc* de taille 512 il aurait été possible d'en avoir 1024.
- ⇨ Plus un *block* contient de threads, plus il utilise de registres, or cette quantité est limitée !

c. Les fonctions, variables et routines principales

Les types de fonctions

__host__ fonction exécutée sur le CPU (par défaut).
__global__ fonction exécutée sur le GPU mais appelée par le CPU.
__device__ fonction exécutée sur le GPU et appelée depuis une fonction GPU.

Les restrictions

__global__ Appelable de l'hôte, exécutée par le GPU, ne peux pas avoir de récursivité, de variable statique, de liste variable de paramètre, ne retourne rien. Incompatible avec **__device__.**
__device__ Appelable du GPU, exécutée sur le GPU, récursivité possible. Incompatible avec **__global__.**

Les variables

__device__ int variable Variable dans la mémoire global du GPU (lente, valide pendant la durée du programme), accédée en interne par les threads, est allouée par le CPU.
__shared__ int variable Variable dans la mémoire partagée du bloc, commune aux threads d'un même bloc, rapide, taille réduite, vie limitée au bloc.
__device__ int variable Variable dans la mémoire constante, visible par tous les threads.
Attention, si rien n'est spécifié, la variable ira dans le registre qui est une mémoire rapide associée au thread.

Les principales fonctions de bases

cudaThreadSynchronize()	bloque jusqu'à ce que le **device** ait terminé les tâches demandées précédemment
cudaChooseDevice()	retourne le **device** qui correspond aux propriétés spécifiées
cudaGetDevice()	retourne le **device** utilisé actuellement
cudaGetDeviceCount()	retourne le nombre de **device** capable de faire du GPGPU
cudaGetDeviceProperties()	retourne les informations concernant le **device**
cudaMalloc()	alloue un objet dans la mémoire globale du **device**.
cudaFree()	libère l'objet de la mémoire globale du **device**
cudaMemcpy()	copie des données de l'hôte vers le **device**.

Les Flux (streams)

Un flux est une source de données, comme un fichier, la mémoire ou le réseau. Dans CUDA, il ne s'agit pas exactement de la même chose.

Pour faciliter les exécutions concurrentes entre hôte et périphérique, certaines fonctions sont asynchrones, dans le runtime (la partie de CUDA que nous apprenons) : l'application récupère le contrôle avant que les calculs soient entièrement effectués. Ainsi, il arrive fréquemment que plusieurs de ces fonctions soient en cours d'exécution en même temps : elles sont concurrentes.
Ces fonctions sont les suivantes.

- ***cuLaunchGrid()*** ;
- ***cuLaunchGridAsync()*** ;
- Les fonctions de copie suffixées **Async** ;
- Les copies du périphérique vers le périphérique ;
- Les appels de **kernels** avec des fonctions **__global__.**

Les applications s'occupent de la concurrence *via* des *flux*. Un flux est une séquence d'instructions, qui doivent s'exécuter dans un certain ordre. D'un autre côté, des flux peuvent arrêter leur exécution pour un autre flux.

Un flux est défini en créant un objet flux et en le spécifiant à un appel de **kernel,** ou à une copie de mémoire. Par exemple, ce code crée deux flux.

```
cudaStream_t stream[2];
for (int i = 0 ; i < 2 ; ++i)
   cudaStreamCreate(& stream[i] );
```

Chacun de ces flux est défini, par ce code, comme une séquence d'une copie de l'hôte au périphérique, d'un appel de **kernel** et d'une copie du périphérique vers l'hôte.

```
for (int i = 0 ; i < 2 ; ++i)
   cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0 ; i < 2 ; ++i)
   mykernel <<< 100, 512, 0, stream[i] >>>   (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0 ; i < 2 ; ++i)
   cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost, stream[i]);
cudaThreadSynchronize();
```

Chaque flux copie sa portion du tableau hostPtr vers inputDevPtr dans la mémoire du périphérique, traite ce dernier tableau et copie le résultat outputDevPtr sur l'hôte, dans la partie correspondante de hostPtr.

cudaThreadSynchronize() est appelée à la fin pour s'assurer que tous les flux ont fini de s'exécuter avant d'essayer d'utiliser leur résultat.
cudaStreamSynchronize() peut être appelée pour synchroniser un flux précis, avec tous les autres flux qui continuent leur exécution normale.
cudaStreamDestroy() est utilisée pour la destruction d'un flux.
cudaStreamQuery() permet de vérifier que toutes les opérations du flux déjà envoyées soient bien effectuées.

Tout appel de kernel ou opération de mémoire sans le paramètre de flux ne commence qu'à la fin des autres opérations. Ces opérations sont affectées au flux 0.

2. La programmation avec Optix

On a vu qu'**Optix** est donc environnement dédié au raytracing. Il utilise ces propres fonctions pour les initialisations de rayons, les intersections de rayons, les ombres... Afin de faciliter le développement d'application, Optix met en avant ces possibilités sous forme d'appel de fonctions programmées, ou modules, en CUDA C. Mais Optix dispose aussi de fonctions dynamiques et de variables qui lui sont propres.
Toutes ces solutions permettent la programmation d'une application puissante mais en gardant un code source relativement simple et court.

Dans la suite de cette annexe, je vais parler des méthodes de programmation sous Optix. La jeunesse de cette technologie et le peu de ressources sont un problème très important lorsque l'on veut apprendre à se servir de cet environnement. J'expliquerai donc certaines notions importantes. Mais tout d'abord voici la liste des objets utilisés dans Optix par le Host Api pour contrôler le raytracer. Car, comme on l'a vu, il faut faire la différence entre le Host Api écrit en C++ et le raytracer exécuté sur GPU par CUDA / OPTIX.

- **Context**: An instance of a running OptiX engine
- **Program**: A CUDA C function, compiled to NVIDIA's PTX virtual assembly language
- **Variable**: A name used to pass data from C to OptiX programs
- **Buffer**: A multidimensional array that can be bound to a variable
- **TextureSampler**: One or more buffers bound with an interpolation mechanism
- **Geometry**: One or more primitives that a ray can be intersected with, suchas triangles or other user-defined types
- **Material**: A set of programs executed when a ray intersects with the closest primitive or potentially closest primitive.
- **GeometryInstance:** A binding between Geometry and Material objects.
- **Group**: A set of objects arranged in a hierarchy
- **GeometryGroup**: A set of GeometryInstance objects
- **Transform**: A hierarchy node that geometrically transforms rays, so as to transform the geometric objects
- **Selector:** A programmable hierarchy node that selects which children to traverse
- **Acceleration:** An acceleration structure object that can be bound to a hierarchy node

Le "Raytracer pipeline" mis en place par Optix contient plusieurs composants programmables utilisés et exécutés par le GPU à des points spécifiques durant l'exécution d'un raytracer:

- **Ray Generation**: The entry point into the ray tracing pipeline, invoked by the system in parallel for each pixel, sample, or other user-defined work assignment
- **Exception**: Exception handler, invoked for conditions such as stack overflow and other errors
- **Closest Hit**: Called when a traced ray finds the closest intersection point, such as for material shading
- **Any Hit**: Called when a traced ray finds a new potentially closest intersection point, such as for shadow computation
- **Intersection**: Implements a ray-primitive intersection test, invoked during traversal
- **Bounding Box**: Computes a primitive's world space bounding box, called when the system builds a new acceleration structure overthe geometry
- **Miss**: Called when a traced ray misses all scene geometry
- **Visit:** Called during traversal of a Selector node to determine the children a ray will traverse

Voici maintenant quelques notions importantes à avoir en tête pour comprendre la programmation avec Optix.

Buffers

Optix utilise des **buffers** pour passer les données du host au device. Ils sont créés avant le lancement d'un context et dirigent le flux de données entre les deux parties du programme.

Context

Dans Optix le **context** est une interface de contrôle pour configurer et diviser les étapes du moteur de rendu raytracing. Il est créé avec la fonction *rtContextCreat.* Ces objets encapsulent toutes les ressources disponibles dans Optix (textures, géométries, paramètres d'utilisation,...). La destruction de cet objet par la fonction *rtContextdestroy* va nettoyer toutes les ressources utilisées et invalider toutes celles créées par lui.
Lancer plusieurs **context** en même temps est inutile car un **context** peut gérer plusieurs hardwares différents.

Entry Points

Chaque **context** peut avoir plusieurs **entry points** pour le calcul. Chaque point d'entrée est associé à un seul programme de génération de rayon. Ceci permet d'avoir la possibilité de changer d'algorithme de rendu ou de faire du rendu "multi pass".

Ray Types

Dans Optix, il y a différents types de rayons (*tableaux ci-contre*) selon ce que l'on veut faire. On fait la distinction entre les rayons générés pour trouver la couleur et ceux utilisés pour déterminer la visibilité d'une source de lumière (*shadows ray*). Cette séparation offre plus de modularité et plus de performance.
Pour sauvegarder le résultat de couleur ou le facteur d'atténuation (ou autre,...), on utilise un "*ray payload*" qui est une structure de données associée à chaque rayon.

Ray Type Purpose	Payload	Closest Hit	Any Hit	Miss
Radiance	`RadiancePL`	Compute color, keep track of recursion depth	n/a	Environment map lookup
Shadow	`ShadowPL`	n/a	Compute shadow attenuation and terminate ray if opaque	n/a

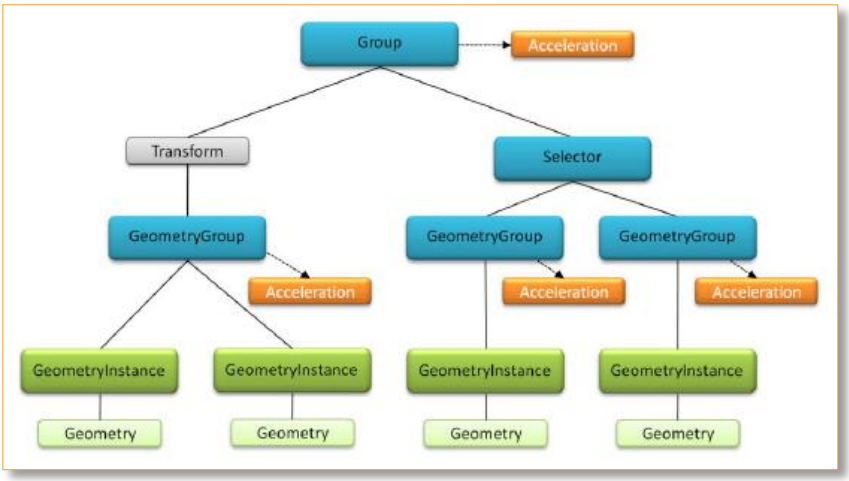
Les différents types de rayons dans Optix (source Nvidia).

Variables

Optix utilise des **variables** pour passer des paramètres dans les paramètres de configurations des programmes. Chaque variable est liée a un context, une géométrie, une sélection, un programme... Elle est visible par l'objet à laquelle elle est liée et à ces enfants. Cependant les variables liées à un context sont visibles par tous les autres objets.

Graph Nodes

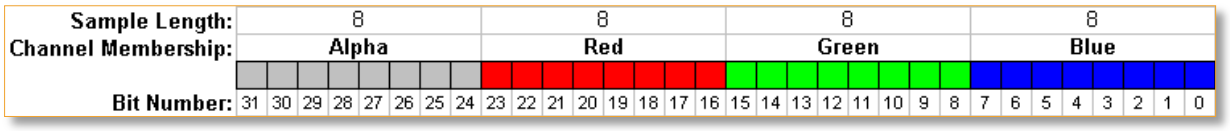
Quand un rayon est tracé avec la fonction *rtTrace*, un noeud est créé qui spécifie que l'on est en haut de l'arbre. L'application host génère cet arbre en assemblant un certain nombre de noeuds différents fournit par l'API Optix. L'architecture de ces arbres est la suivante : chaque noeud décrit un objet géométrique en haut et des objets en bas.



Hiérarchie des objets dans Optix (source Nvidia).

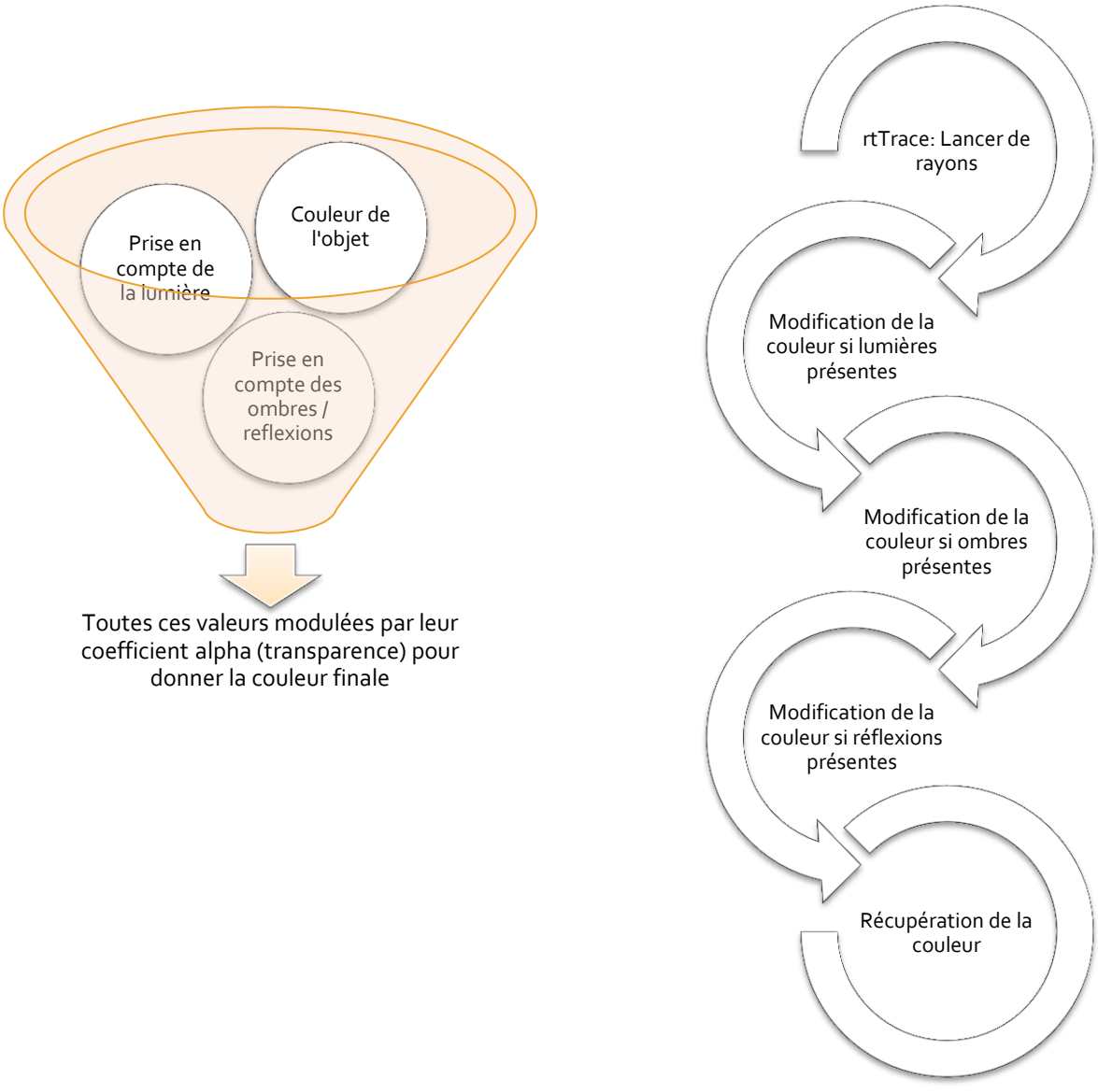
Les couleurs

Dans Optix, la couleur est gérée différemment. Alors que pour une image classique la couleur d'un point est définie par un couple de variables RGB (donc trois valeurs distinctes de type float), sous Optix ces valeurs sont contenues dans une structure de type uchar4 (donc 4 unsigned character: valeur de 0 à 255) pour R, G, B et A (Alpha). Cela permet de faire passer la couleur pour une texture affichable par OpenGL.
Dans le système **RGBA** (*red green blue alpha*, *image ci-dessous*), on rajoute une valeur alpha pour rajouter des effets de "alpha blending" (association complexe de deux couleurs afin de rendre possible des effets de transparence) et de "alpha compositing"(processus de combination d'une image et d'un arrière-plan qui permet de créer des transparences).



Programmation de la couleur en RGBA.

Cette gestion des couleurs permet l'utilisation d'effet pour les réflexions, les ombres, les diffusions... Voici comment elles sont calculées :


Prise en compte de la lumière
Couleur de l'objet
Prise en compte des ombres / reflexions
Toutes ces valeurs modulées par leur coefficient alpha (transparence) pour donner la couleur finale
rtTrace: Lancer de rayons
Modification de la couleur si lumières présentes
Modification de la couleur si ombres présentes
Modification de la couleur si réflexions présentes
Récupération de la couleur

Calculs des couleurs dans Optix.

3. Les différents types de fichiers

a. Les fichiers OBJ (article issu de wikipédia)

OBJ ou .obj est un format de fichier qui définit des géométries 3D. Il a été inventé par Wavefront Technologies pour son logiciel Advanced Visualizer. Ce type de fichier est ouvert et a donc été repris par de nombreux logiciels de modélisation 3D. C'est devenu un format de fichier universel.
Le format OBJ est un simple format de données qui représente une géométrie 3D seule en donnant la position de chaque **vertex,** la position de chaque texture et les faces qui font que chaque polygone est défini comme une liste de vertex, textures. Les vertex sont stockés dans un ordre antihoraire par défaut.

Les vertex: un index valide de vertex commence à un et concorde avec le dernier élément de la liste précédente. Chaque face peut contenir plus de trois vertex.

```
p v1 v2 v3 v4 ...
```

Les normales: donne la normale à la facette en x,y,z .

```
vn 0.707 0.000 0.707 ...
```

Les coordonnées vertex/texture: on peut définir, grâce aux coordonnées de texture, la texture d'une face. Un index de coordonnées de textures valide part de 1 et concorde avec le dernier élément de la liste précédente. Chaque face peut contenir plus de trois éléments.

```
vt v1/vt1 v2/vt2 v3/vt3 ...
```

Définition des faces: Les faces sont définies en utilisant la liste des vertex, textures, normal. Les polygones comme les quadrilatères peuvent utiliser plus de trois vertex, textures, normal.

```
f 1 2 3 ...
```

Les coordonnées vertex/texture/normal: on peut définir, grâce aux coordonnées des normales, le vecteur normal lorsque l'on crée une face. Un index de coordonnées de normales valide part de 1 et concorde avec le dernier élément de la liste précédente. Chaque face peut contenir plus de trois éléments.

```
f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3 ...
```

Les coordonnées vertex/normal: comme précédemment mais sans les textures.

```
f v1//vn1 v2//vn2 v3//vn3 ...
```

Les matériaux et autres options: Les matériaux qui décrivent l'aspect des polygones sont sauvegardés dans un fichier externe .mtl. Ce fichier contient un ou plusieurs matériaux. Voici comme il est annoncé dans le fichier OBJ:

```
mtllib [external .mtl file name]...
```

Les matériaux, groupes de polygones ou objets sont spécifiés de la manière suivante:

```
o [object name]...
g [group name]...
usemtl [material name]...
```

Construction d'un fichier OBJ

La première ligne est un commentaire: # this is a comment
Ensuite on a un enchainement de vertex, vertex normal, faces et cela une ou plusieurs fois:

```
# List of Vertices, with (x,y,z[,w]) coordinates, w is optional and defaults to 1.0.
   v 0.123 0.234 0.345 1.0...
# Texture coordinates, in (u,v,w) coordinates, v and w are optional and default to 0.
   vt 0.500 -1.352 [0.234]...
# Normals in (x,y,z) form; normals might not be unit.
   vn 0.707 0.000 0.707...
# Face Definitions
   f 1 2 3
   f 3/1 4/2 5/3
   f 6/4/1 3/5/3 7/6/5...
```

Les fichiers .mtl

Le format **MTL** (**Material Template Library**, ou **Material Library file**) est un standard définit par Wavefront Technologies en complément du format OBJ. Le fichier **.mtl**, est un fichier au format ASCII (texte) qui contient la définition d'un ou de plusieurs matériaux d'un objet 3D. Chacune de ses définitions incluent la coloration, la texture et les paramètres de réflexion optique de chacun de ces matériaux.

On peut définir plusieurs matériaux dans un fichier .mtl, chaque matériau est précédé de la commande: `newmtl`

```
# define a material named 'Colored'
newmtl Colored
```

La couleur ambiante est définie par `Ka`. La couleur est définie en RGB avec des valeurs allant de 0 à 1.

```
Ka 1.000 1.000 1.000     # white
```

De la même façon la couleur diffuse est définie par `Kd`.

```
Kd 1.000 1.000 1.000     # white
```

La couleur spéculaire est définie par `Ks`, et on utilise le coefficient spéculaire `Ns`.

```
Ks 0.000 0.000 0.000     # black (off)
Ns 10.000                # ranges between 0 and 1000
```

De nombreux modèles d'illuminations sont possibles pour les matériaux:

```
0. Color on and Ambient off
1. Color on and Ambient on
2. Highlight on
3. Reflection on and Ray trace on
```

b. Les fichiers BDD (article technique de l'Onera)

C'est un format de type texte piloté par des mots clés ou balises. La portée d'une balise est limitée par l'entrée d'une autre balise. En général, la fin des données liées à une balise est séparée de la balise suivante par une ligne blanche, mais ce n'est pas obligatoire. Lorsqu'une ligne comporte plusieurs données, celles-ci sont séparées par au moins un espace blanc et/ou une tabulation.

Ce format est multi objet. Chaque objet possède au moins un tableau de points et un tableau de facettes. Il peut avoir en option un tableau de normales aux facettes, un tableau de normales aux sommets, un tableau de numéros de groupes, un tableau de numéros de matériaux et éventuellement une matrice de transformation associée à un objet. Cette matrice sert à modifier l'échelle de l'objet, à le déplacer ou à en effectuer une rotation.

Le mot clé **<OBJET>** est suivi, sur la même ligne, d'un numéro d'objet et d'un nom d'objet. Chaque apparition d'une balise **<OBJET>** défini un nouvel objet. Il est préférable que deux objets n'aient pas le même numéro ni le même nom.
Le mot clé **<SOMMET>** est suivi sur la même ligne du nombre de points qui forment l'objet. Les lignes suivantes comportent d'abord le numéro de point (entier) puis les 3 coordonnées (format flottant) en x, y et z du point. Le format est libre, séparé par des espaces et/ou des tabulations. Il doit y avoir autant de lignes que de nombre de points déclarés avec le mot-clé. La numérotation commence à 1.
Le mot clé **<FACE>** permet de définir les facettes d'un objet. Le mot clé est suivi du nombre de facettes. Les lignes suivantes comportent d'abord le numéro de la facette, suivi du nombre de sommets de la facette et enfin, pour chaque sommet, le numéro d'indice des sommets dans le tableau des points. La numérotation des facettes commence à 1.
Seuls les mots clés **<OBJET>**, **<SOMMET>** et **<FACE>** sont obligatoires.

Le mot clé **<NORMALE>** permet d'entrer les normales aux barycentres des facettes.
Le mot clé **<VECTEUR>** permet d'entrer un tableau de normales aux sommets. Il est suivi, sur la même ligne, du nombre de normales aux sommets. Ce nombre peut être égal au nombre de sommets, mais ce n'est pas une obligation.
Le mot clé **<LUMINANCE>** (idem en version anglaise) permet de définir les normales à affecter à chaque sommet de facette. Il est suivi du nombre de facettes pour lesquelles on entre des normales aux sommets.

Les lignes commençant par un dièse sont considérées comme des commentaires.

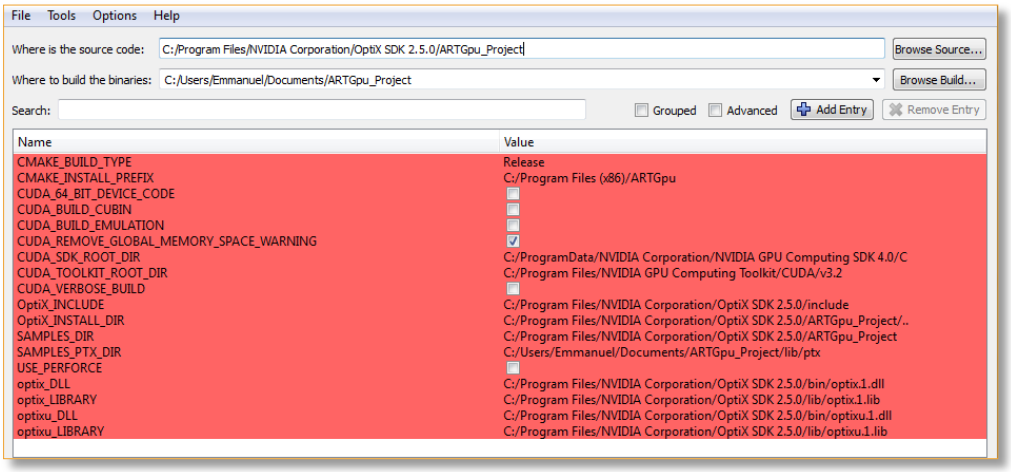
4. Manuel d'utilisation ARTGpu

a. Installation version développeur

Pour installer le logiciel, il faut suivre plusieurs étapes. Il faut au préalable avoir installé Visual Studio (2008 de préférence), le toolkit CUDA, le SDK OPTIX et MinGW et Cmake.
Télécharger Cmake: http://www.cmake.org/cmake/resources/software.html

Copier le dossier *ART_GPU* dans le dossier par défaut d'installation d'Optix:
C:\Program Files\NVIDIA Corporation\OptiX SDK 2.5.0

Lancez l'application *cmake-gui* et définir le chemin de vos sources (*C:\Program Files\NVIDIA Corporation\OptiX SDK 2.5.0\ARTGpu_Project*) et le dossier destination qui contiendra tous les fichiers de compilations, exécutables,... (par exemple *C:\Users\X\Documents\ARTGpu_Project*).
Cliquez sur ***configurer*** et choisir dans la nouvelle fenêtre votre IDE (exemple : Visual Studio 2008). **ATTENTION** bien choisir x64 si l'IDE fonctionne en 64bits et x32 s'il fonctionne en 32 bits. Et cliquer ***OK***. Cmake va mouliner et des informations vont apparaître sur votre écran.



Fenêtre de configuration Cmake.

Cliquez une nouvelle fois sur ***configurer*** puis sur ***générer***. ARTGpu est prêt à être compilé.

Rendez-vous dans le dossier destination que vous avez défini précédemment et ouvrez le fichier .sln qui vous ouvrira VS2008. Faite un build du projet: ***Build/Build Solution.*** Maintenant que le build est terminé, les exécutables de ARTGpu sont disponibles.

Pour finir il faut modifier le path dans le script perl ***Start.pl*** :
my $chemin ="C:/Users/xxxxxxxxxx/Documents/ARTGpu_Final/bin/Debug/"; Remplacer le chemin par le vôtre.

Les scripts perl et l'exécutable ***nconvert.exe*** doivent être dans le répertoire ***Debug*** de la solution donc dans le même répertoire que les exécutables. Et ensuite lancer le script (vous devez utiliser Mingw): ***perl Start.pl***

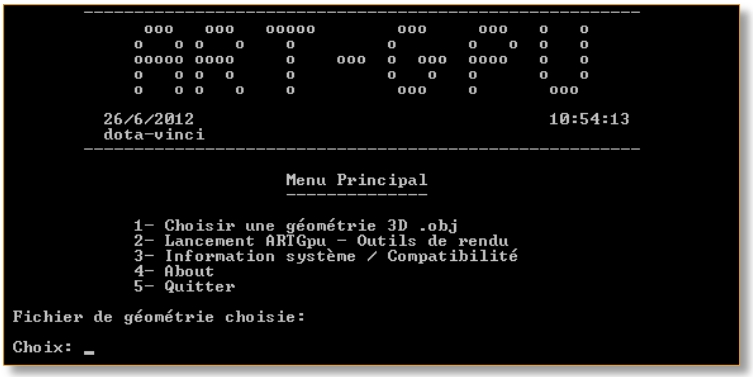
Voilà ARTGpu est utilisable !

b. Installation version utilisateur

Récupérez le fichier ***ARTGpu-setpu.exe*** et installez-le comme n'importe quel logiciel windows. La version utilisateur ne permet pas de modifier le code source. On ne peut pas non plus choisir le répertoire d'installation pour des raisons de fonctionnement dans le code source.

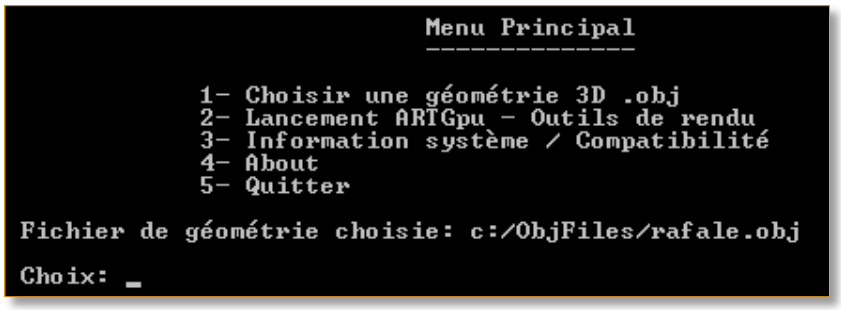
c. Fonctionnement détaillé de ARTGpu

Lancement du script perl grâce à MinGW : ***perl Start.pl*** (ou grâce au raccourci : version utilisateur)



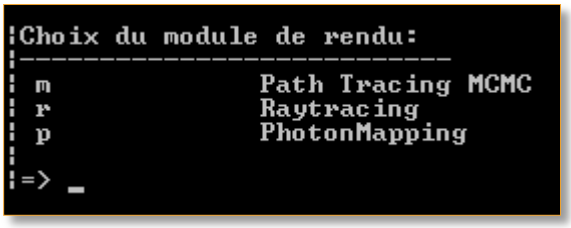
Vous vous trouvez dans le menu principal de ARTGpu. Vous avez donc plusieurs choix : entrer le chemin vers le fichier obj, lancer le rendu de la géométrie (il faut bien sûr en avoir rentrée une avant), tester la compatibilité de son poste de travail, afficher les informations de ARTGpu et enfin quitter.
Si on veut effectuer un rendu, la première chose à faire et de rentrer un fichier .obj (commande 1 dans le menu). Entrez le chemin vers le fichier obj (en utilisant la notation linux: c:/.../.../...) puis ***Entrée.***

Après avoir entré la géométrie, vous verrez votre choix s'afficher dans le menu :



Vous pouvez maintenant lancer un rendu avec la géométrie sélectionnée, en utilisant la commande 2 du menu.

Choisir le module de rendu que l'on veut utiliser : raytracing, photon mapping ou path tracing.



Vous avez maintenant trois choix (ces choix sont les mêmes pour les trois modules de rendu) :

- ***miss:*** utilisation d'une couleur d'arrière-plan unie que vous rentrerez vous même.
- ***env:*** utilisation d'une image type hdr qui fera office d'environnement map de la scène. Le programme vous demandera de rentrer son chemin juste après et si une image est déjà présente il vous demande si vous voulez l'utiliser ou non.
- ***:*** juste appuyer sur ***Entrée*** sans choisir d'option chargera les paramètres par défaut.

```
¦Background options:
¦--------------------------
¦ miss          Couleur d'arrière plan uniforme
¦ env           Environnement Mapping
¦
¦=>
```

Viens ensuite le script matériaux.pl qui s'exécute et qui va soit vous afficher les informations du mtl existant et faire les conversions des textures si nécessaire, soit créer un fichier mtl grâce à un fichier référence Data_BRDF fournit par Crira.

```
¦Gestion des fichiers materiaux:
¦---------------------------
¦Le fichier rafale.mtl existe!
¦Aucune conversion d'image nécessaire!
¦Il y a 3 materiaux dans le fichier.
```

Un choix vous est ensuite demandé (si vous avez choisi le module Raytracing) : Utiliser les jets ou pas ! Si vous choisissez de les utiliser, le script jets.pl va convertir le fichier jet_SDM (format bdd) fournit par Crira au format obj. Si vous choisissez dans les options de rendu l'option jets <n> (avec le nombre de jets à afficher, ARTGpu va intégrer les jets dans la scène).

```
¦Gestion des jets:
¦---------------------------
¦Voulez vous genérer les fichiers de jets? (y/n)
¦
¦=>
```

Ensuite vous avez deux types d'options : les options ***Opengl*** qui touchent la fenêtre windows et les options de rendu qui vont modifier la façon dont ARTGpu fait les calculs.

```
        ------------------------------------------------------
         ARTGpu         Module Raytracing
        ------------------------------------------------------

¦Gestion des jets:
¦---------------------------
¦Voulez vous genérer les fichiers de jets? (y/n)
¦
¦=> n
¦Gestion de jets non choisi!

¦OpenGl options:
¦---------------------------
¦ -d  ¦ -d=<width>x<height>  Set image dimensions
¦ -D  ¦ -D=<num_devices>     Set desired number of GPUs

¦ARTGpu options:
¦---------------------------
¦ ao                 Use progressive Ambient Occlusion shader
¦ aop                Use progressive Ambient Occlusion and Phong shader
¦    m   ¦ m <n>      Multiplier for the number of AO samples
¦    r   ¦ r <scale>  Scale ambient occlusion radius
¦ ns                 Use normal shader (show illumination map)
¦ ds                 Use one bounce Diffuse shader
¦ fr                 Use Fresnel shader (fresnel reflexion)
¦ ph                 Use Phong shader (ambient, shadows, reflexion)
¦ aa                 Use subpixel jittering to perform antialiasing
¦ oc                 Use orthographic camera (cannot use AO mode with ortho)
¦ l     ¦ l <scale>  Scale lights by constant factor
¦ kd                 Kd Tree acceleration
¦ lbvh               lbvh acceleration
¦ bvh                bvh acceleration
¦ t     ¦ t <n>      Progressive Render Time (par defaut 10s)
¦ axe                Display x y z axes
¦ jets               Utilisation du fichier jet pour l'avion

Paramètres (chacun séparé par un espace)=> _
```

Pour rentrer une option, vous tapez juste la commande qui vous convient (exemple: aop pour ambient occlusion and phong). Vous pouvez taper plusieurs commandes à la suite du moment qu'elles sont séparées par un espace. Il faut comprendre que tous ces paramètres ainsi que ceux tapés précédemment sont passés à ARTGpu par l'intermédiaire des tableaux d'arguments argv / argc. Exemple de ligne de paramètres: ***aop aa m 4 t 20***

ARTGpu se lance maintenant que la phase de préparation et de paramétrage est finit (sauf si vous avez choisi ***miss***, ARTGpu vous demandera la couleur (format float RGB) d'arrière-plan). Si vous avez choisi le module photon mapping, ARTGpu vous demandera les paramètres de la lumière à utiliser. Le premier écran présente les options propres à la fenêtre windows qui vient de s'ouvrir avec les interactions souris / clavier possibles.

```
-------------- Lancement de ARTGpu --------------

Mouse interaction:
  left mouse            Camera Rotate/Orbit
  middle mouse          Camera Pan/Truck
  right mouse           Camera Dolly
  right mouse + shift   Camera FOV

Keyboard interaction:
  q Quit
  f Toggle full screen
  b Start/stop a benchmark
  d Toggle frame rate display
  s Save a frame to 'SavePicture.ppm'
  m Toggle memory usage printing
  c Print camera pose
  l Print Sun pose (x X | y Y | z Z)

Aucun parametre de rendu entre!

Aucun parametre d'environnement entre!
```

Le deuxième écran est là pour informations en donnant le temps de chargement, compilation, création de l'arbre…

```
Time to load geometry: 0.86546 s.
Time to compile kernel: 0.690303 s.
Time to build AS      : 1.57013 s.
```

En quittant la fenêtre OpenGl, ARTGpu se ferme et va automatiquement vérifier si vous avez effectué des sauvegardes d'images et les convertir / déplacer si il en trouve.

```
|-------------- Convertion des images --------------
|Nombre d'images: 0
```

Pour finir, ARTGpu vous propose de relancer le calcul.

```
|-------------- Fermeture de ARTGpu --------------
|Voulez vous relancer un calcul (y/n): y
```

Si vous choisissez non, ARTGpu reviendra au menu principal.

Intéressons-nous maintenant au module information et compatibilité (choix 3 dans le menu):

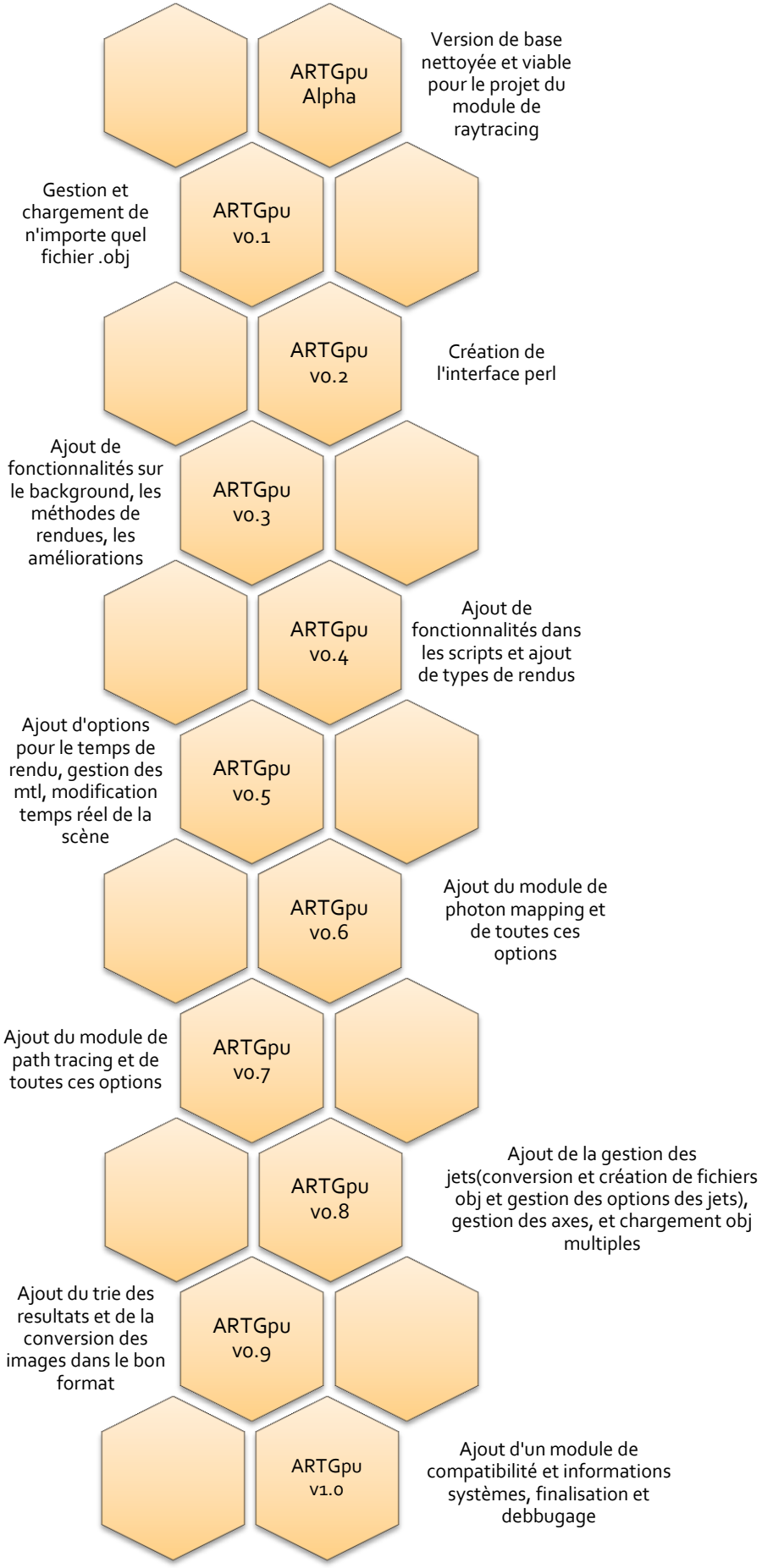
```
        ------------------------------------------------------
         ARTGpu          Informations Systèmes
        ------------------------------------------------------

#----- Environnement et Cpu -----
|
|  Poste de travail:          DOTA-VINCI
|  Systeme d'exploitation:    Microsoft Windows 7 Professionnel
|  Type du systeme:           x64-based PC
|  Nombre(s) Processeur(s):   1
|       Détails:  Intel64 Family 6 Model 26 Stepping 5 GenuineIntel ~3193 MHz
|  Memoire RAM totale:        16 374 Mo
|  Memoire RAM disponible:    13 755 Mo
|
```

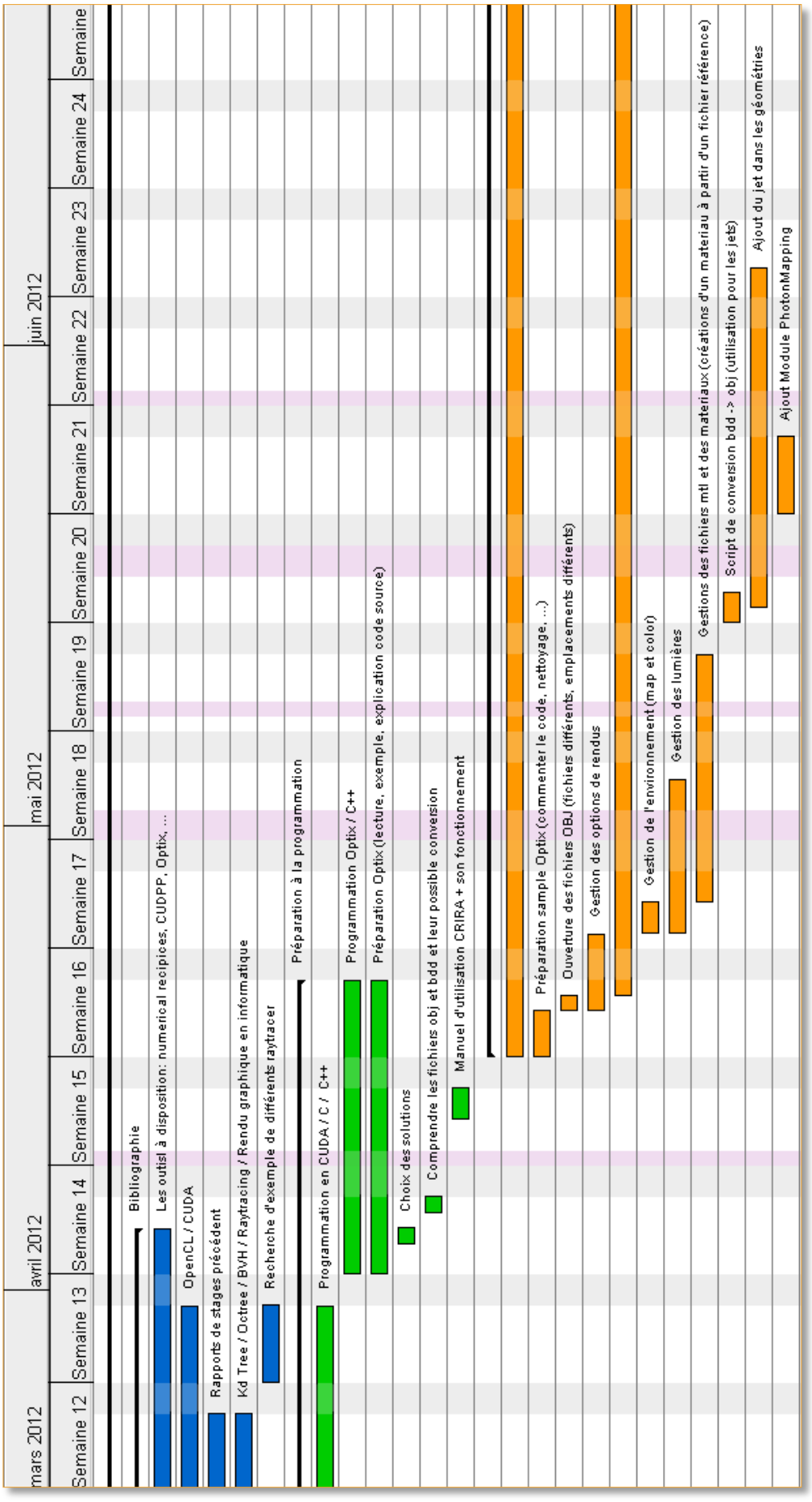
…

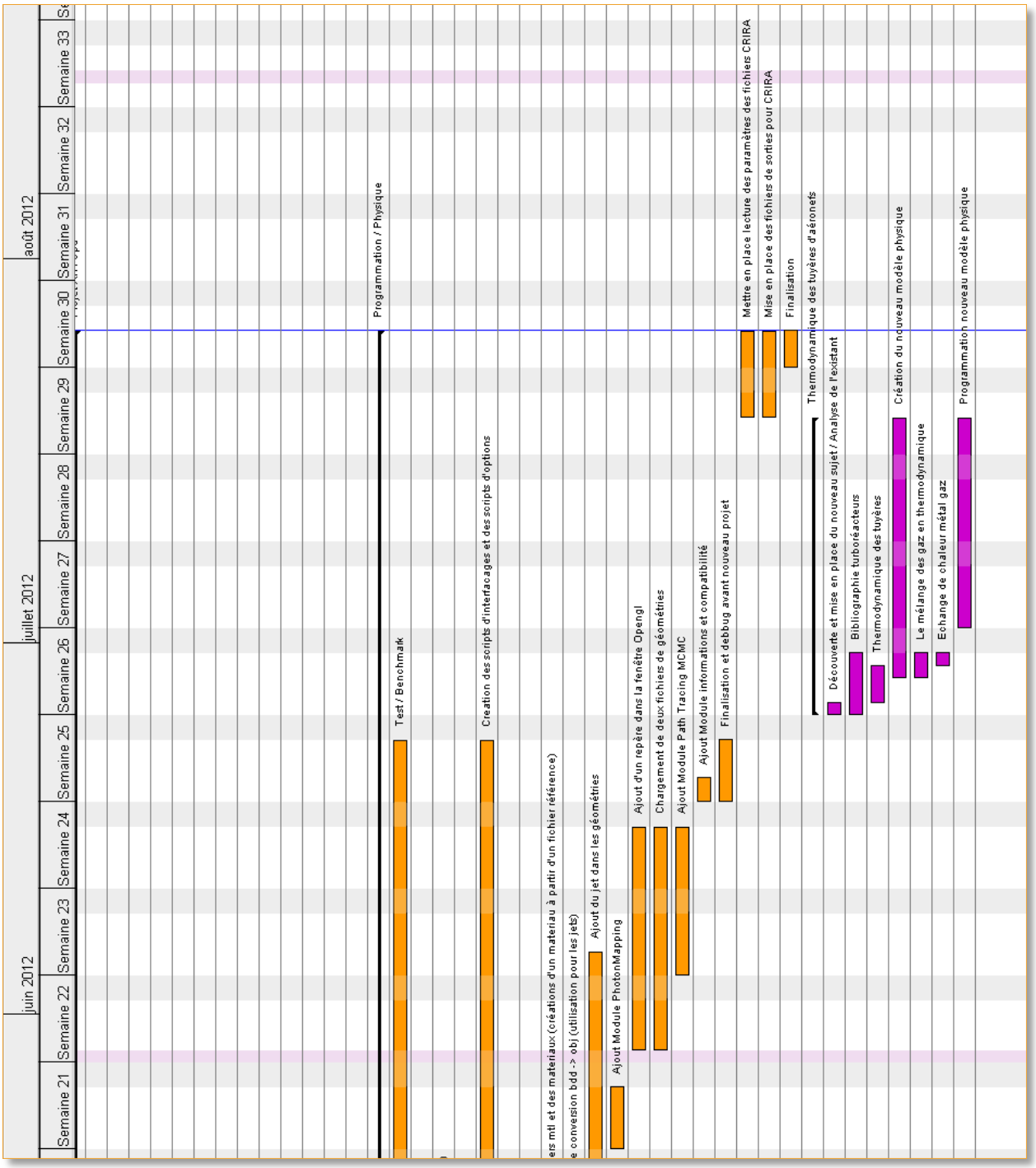
```
#----- Compatibilite -----
|
|  Votre Ordinateur est compatible avec ARTGpu
|
```

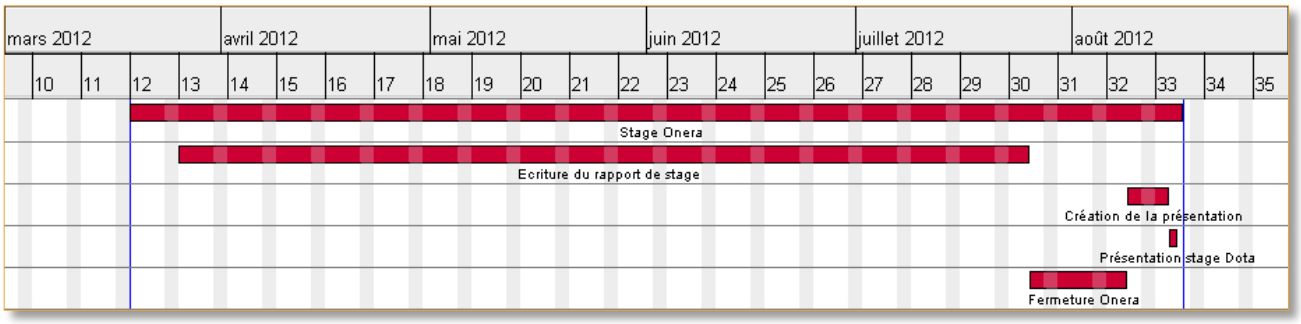
d. Historique des versions


ARTGpu Alpha
Version de base nettoyée et viable pour le projet du module de raytracing
Gestion et chargement de n'importe quel fichier .obj
ARTGpu v0.1
ARTGpu v0.2
Création de l'interface perl
Ajout de fonctionnalités sur le background, les méthodes de rendues, les améliorations
ARTGpu v0.3
ARTGpu v0.4
Ajout de fonctionnalités dans les scripts et ajout de types de rendus
Ajout d'options pour le temps de rendu, gestion des mtl, modification temps réel de la scène
ARTGpu v0.5
ARTGpu v0.6
Ajout du module de photon mapping et de toutes ces options
Ajout du module de path tracing et de toutes ces options
ARTGpu v0.7
ARTGpu v0.8
Ajout de la gestion des jets(conversion et création de fichiers obj et gestion des options des jets), gestion des axes, et chargement obj multiples
Ajout du trie des resultats et de la conversion des images dans le bon format
ARTGpu v0.9
ARTGpu v1.0
Ajout d'un module de compatibilité et informations systèmes, finalisation et debbugage

5. Diagramme de Gantt


mars 2012
avril 2012
mai 2012
juin 2012
Semaine 12
Semaine 13
Semaine 14
Semaine 15
Semaine 16
Semaine 17
Semaine 18
Semaine 19
Semaine 20
Semaine 21
Semaine 22
Semaine 23
Semaine 24
Semaine
Bibliographie
Les outils à disposition: numerical recipies, CUDPP, Optix, ...
OpenCL / CUDA
Rapports de stages précédent
Kd Tree / Octree / BVH / Raytracing / Rendu graphique en informatique
Recherche d'exemple de différents raytracer
Préparation à la programmation
Programmation en CUDA / C / C++
Programmation Optix / C++
Préparation Optix (lecture, exemple, explication code source)
Choix des solutions
Comprendre les fichiers obj et bdd et leur possible conversion
Manuel d'utilisation CRIRA + son fonctionnement
Préparation sample Optix (commenter le code, nettoyage, ...)
Ouverture des fichiers OBJ (fichiers différents, emplacements différents)
Gestion des options de rendus
Gestion de l'environnement (map et color)
Gestion des lumières
Gestions des fichiers mtl et des materiaux (créations d'un materiau à partir d'un fichier référence)
Script de conversion bdd -> obj (utilisation pour les jets)
Ajout du jet dans les géométries
Ajout Module PhotonMapping


juin 2012
juillet 2012
août 2012
Semaine 21
Semaine 22
Semaine 23
Semaine 24
Semaine 25
Semaine 26
Semaine 27
Semaine 28
Semaine 29
Semaine 30
Semaine 31
Semaine 32
Semaine 33
Programmation / Physique
Test / Benchmark
Creation des scripts d'interfacages et des scripts d'options
ers mtl et des materiaux (créations d'un materiau à partir d'un fichier référence)
e conversion bdd -> obj (utilisation pour les jets)
Ajout du jet dans les géométries
Ajout Module PhotonMapping
Ajout d'un repère dans la fenêtre Opengl
Chargement de deux fichiers de géométries
Ajout Module Path Tracing MCMC
Ajout Module informations et compatibilité
Finalisation et debbug avant nouveau projet
Mettre en place lecture des paramètres des fichiers CRIRA
Mise en place des fichiers de sorties pour CRIRA
Finalisation
Thermodynamique des tuyères d'aéronefs
Découverte et mise en place du nouveau sujet / Analyse de l'existant
Bibliographie turboréacteurs
Thermodynamique des tuyères
Création du nouveau modèle physique
Le mélange des gaz en thermodynamique
Echange de chaleur métal gaz
Programmation nouveau modèle physique


mars 2012
avril 2012
mai 2012
juin 2012
juillet 2012
août 2012
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
Stage Onera
Ecriture du rapport de stage
Création de la présentation
Présentation stage Dota
Fermeture Onera